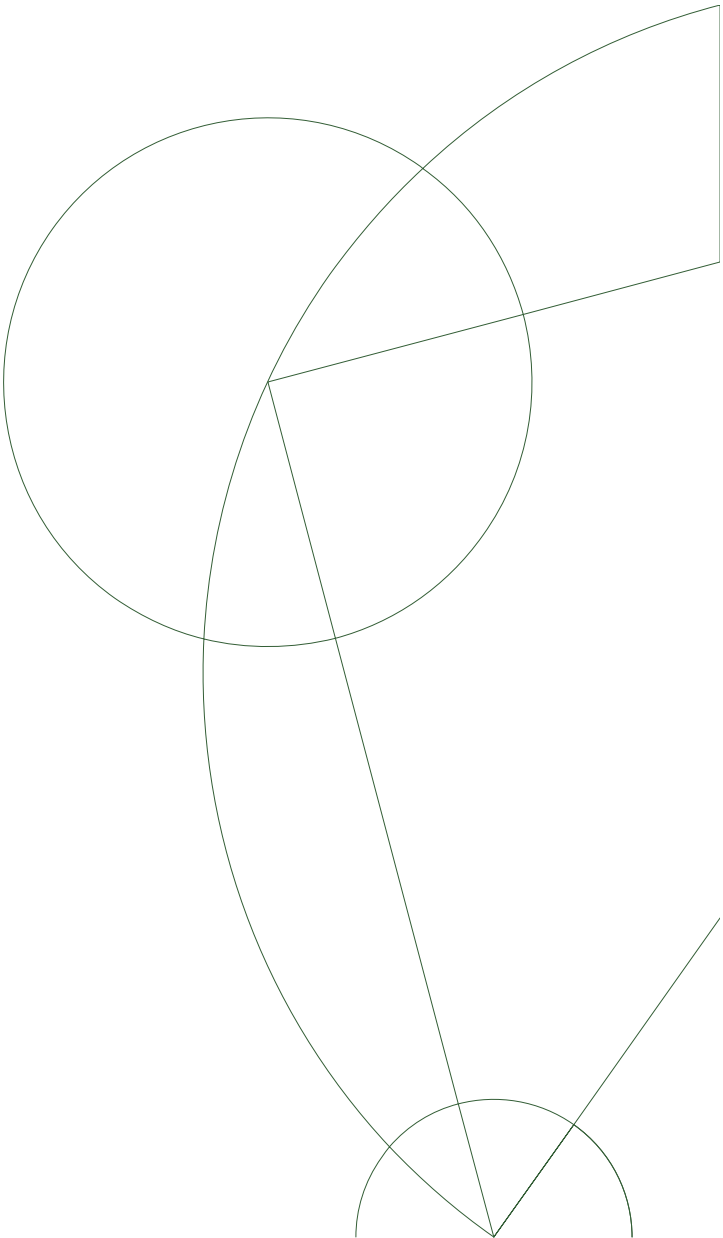
KØBENHAVNS UNIVERSITET

Speciale

Jens Peter Svensson

Design og implementering af programbiblioteket CPH collections



Vejleder: Jyrki Katajainen

Afleveret: februar 2011

Resumé. Specialet omhandler designet og implementeringen af programbiblioteket CPH collections. CPH collections er et program bibliotek der implementeres i Python og som er en delmængde af C++ standard biblioteket fra C++0x specifikationen [2]. I specialet beskrives og begrundes de designvalg, der foretages i designet af CPH collections og de teknikker, der bruges til at implementere biblioteket, vil blive beskrevet. Specialet bliver afsluttet med ideer og forslag til fremtidige udviklere af CPH collections.

Hovedformålet med at implementere C++ standard biblioteket i Python er antagelsen om, at det er lettere at eksperimentere og lære i Python. Derfor burde det være lettere at undervise studerende i algoritmer og datastrukturer i Python med CPH collections end i C++ med C++'s standard bibliotek.

Grundstenen til CPH collections blev lagt i "Algoritmer og datastrukturer" kurset foråret 2010, hvor de studerende blev bedt om at implementere komponenter fremlagt af underviseren. En del af specialet har dermed bestået i at samle komponenter skrevet af personer med forskellig stil i et enkelt sammenhængende bibibliotek. Nogle af løsningerne fra de studerende er ikke med da disse ikke var kompatible eller funktionelle med resten af CPH collections. Der vil ikke i dette speciale blive talt mere om dette.

Abstract. The master thesis is about the design and implementation of the program library CPH collections. CPH collections is a program librabry that is implemented in Python and consists of a subset of the C++ standard library from the C++ specification. The thesis will describe and give reasons for design choices made in the design of the library and the techniques used in the implementation of the library. The thesis will be ended with ideas and suggestions for future developers of CPH collections.

The main purposs with implementing CPH collections is the presumption that it is easier to learn Python than C++ and also that it is easier to experiment in Python. Thus it should be easier to teach students about algorithms and datastructures using Python and CPH collections than using C++ and its standard library.

The initial codebase to CPH collections was made the course "Algoritmer og datastrukturer" spring 2010. In this course the students was assigned to develop components from the coursebook and teacher in Python. A part of this thesis has thus been to collect these different components written by different people with different styles into a unified library. Some of the solutions made by the students is not in the library cause they was incompatible or non funtioning with CPH collections. There will in this thesis not be talked about this further.

Indhold

Kapitel 1

Konklusion

Formålet med CPH collections var, at det skulle kunne bruges til let at lære at skrive algoritmer. Om dette formål opnås ved at implementere C++ standard biblioteket i Python er ikke bevist. Dog kan det antages, at CPH collections kan bruges som en middelvej til at lære at skrive kode i C++, hvor denne middelvej abstraherer væk fra dele af C++-syntaksen i brug af standard biblioteket; specielt væk fra pegere og skabeloner i C++.

Der blev foretaget designvalg for at gøre brugen af CPH collections lettere for dets brugere. Blandt disse er, at alle beholdertyper i biblioteket har en fabrik til at returnere instanser af beholderen. At iteratortyper ikke er bundet eller kan blive sat i en beholder, men, der kan oprettes instanser af iteratortyper ved brug af en fabrik. At algoritmer i CPH collections tager en samling (beholder eller iterator) i stedet for to iteratorer som i C++. Og at alle pakker henter deres indhold ind.

Med beholderfabrikkerne opnås, at brugerne dermed ikke behøver at kende navnene på realisationerne af en beholder, men kun, hvordan en realisation fås fra fabrikken. Dette er dog ikke helt implementeret; brugerne kan kun få en specifik realisation med en kombination af bogstaverne i dennes navn. Det er dog muligt at udvide fabrikkerne til at acceptere andre parametre. En fordel ved fabrikkerne, som ikke blev testede, er, at hvilken realisation der fås fra fabrikken kan ændres ved ændringer i en streng værdi. Det er dermed lettere at ændre realisationen, der bruges fra brugerens side; blot ved ændring i en tekststreng som for eksempel kan skrives i en konfigurationsfil.

Ved adskillelsen af iteratorer fra grænsefladen til beholderne opnås, at andre typer af iteratorer til en beholder kan implementeres og deres brug forstås af brugerne. Med kombinationen af frie iteratorer og iteratorfabrikken kan en ny iteratortype implementeres uden, at det kræver ændringer af eksisterende beholderimplementeringer, og med fabrikken opnås at instansieringen af en iteratortype er den samme som andre iteratortyper. Brugen af frie iteratorer og iteratorfabrikken er dog kun en forbedring, hvis biblioteket udvides med andre iteratortyper. Hvis der kun bruges forlæns- og baglænsiterator, opnås der intet ved den ekstra kompleksitet iteratorfabrikken pålægger i forhold til at kunne få

en iterator fra beholderens grænseflade.

Ved, at algoritmer tager en samling i stedet for to iteratorer, bliver brugen af CPH collections lettere når algoritmen ønskes brugt til alle elementer i en samling, men det fjerner også muligheden for at bruge algoritmen på et udsnit af samlingen. At kunne bruge en algoritme på et udsnit kunne opnås ved implementering af en iterator, der løber mellem to iteratorer eller et specificeret antal skridt.

Valg om, at pakker henter indholdet ind fra deres moduler og lader indholdet være tilgængeligt i pakken, og sammen med et valg om, at beholderfabrikkerne er tilgængelige i bibliotekets hovedpakke, medføre reelt, at brugerne kun behøver at importere bibliotekets hovedpakke for at importere hele CPH collections.

Erfaringen med designvalg er, at desto tidligere de foretages desto mindre skal refaktoreres, men også, at det er svært at gennemskue alle konsekvenserne af et designvalg når det foretages og disse først bliver observeret senere. For eksempel bliver elementer, givet til beholdere i deres oprettelse som en samling, skubbet ind i beholderen i samlingens orden. Effekten af dette designvalg er, at elementerne ligger omvendt i en stak end i alle andre beholdere. Denne forskel er først synlig i modultestene af algoritmer, der skal testes med to forskellige koder afhængig af om beholderen er en stak eller ej.

Andre designvalg blev foretaget for at øge bibliotekets effektivitet med henhold til køretid. Om disse valg er fornuftige kan der stilles spørgsmåltegn ved, da ydelsen af Python-kode kan forbedres på andre måder. For eksempel med værktøjer som *psyco*.

Ved hvert nyt designvalg skal det overvejes om kodebasen og dokumentationen (eksempler) skal refaktoreres. Hvis designvalget ændrer, hvordan biblioteket bruges af brugerne, vil det være en fordel at refaktorere det til hele biblioteket da dette sikrer en ensartet brug, men hvis valget ikke er synligt for brugeren, som for eksempel ydelsesvalg, er det ikke nødvendigt at refaktorere det ind i eksisterende dele af biblioteket.

I designet af CPH collections bliver designmønstre bevidst brugt og deres brug er indikeret i navngivningen af elementer, der bruger dem. Dette medfører, at det er lettere for udviklere at læse og forstå formålet med et kodeelement. For eksempel, hvis udvikleren læser *Brigde* i navnet på en klasse ved han, at denne klasse er en bro og dermed også, hvordan koden til denne klasse skal læses. Brugen af designmønstre betyder, at det er lettere for brugerne at bruge biblioteket, som for eksempel fabrikker og iteratorer, men de kan også bruges til at øge genbrugen af kode, som for eksempel broer, eller til at mindske mængden af kode, som skal skrives, som for eksempel med abstrakte klasser. Implementeringer bør ikke kun skrives for at bruge et designmønster, det er for eksempel spekulativt, hvor fornuftigt det er at implementere en stak som en adaptation af Pythons list, der er en kontinuerlig blok i hukommelsen.

Det blev forsøgt at skrive implementeringer af kodeelementer i `C`. Erfaringen fra dette er, at det er let at implementere de ikke modificerende algoritmer i `C` og svært at implementere beholderne. Det blev ikke forsøgt at implementere modificerende algoritmer i `C`. Eftersom, at alle algoritmerne er baseret på iterering er besparelsen, hvis nogen, ikke stor ved en implementering i `C` da disse

stadig bruger iteratorer implementeret i Python. Derimod er besparelsen stor, hvis en beholder og dens iteratorer implementeres i `C`. (Iteratoren kan gå uden om Python og bruge `C` til at interagere med beholderen.) Ved at implementere i `C` læres dog hverken, hvordan klasser skrives i C++ eller Python. Dermed er fornuften i brug af `C` ikke stor, hvis formålet med biblioteket er at lære at skrive algoritmer.

Der er skrevet dokumentation til alle metoder, funktioner, klasser og moduler i biblioteket. Dette blev gjort i selve kodefilerne, en ofte sagt påstand er at dokumentation skrevet i koden altid er opdateret. Erfaringen fra dette bibliotek er, at dette kun passer, hvis dem, der retter i koden, altid opdaterer dokumentationen samtidig. Endvidere fylder dokumentationen, i det valgte format, ofte mere end selve koden som dokumenteres. At rette fejl fundet i den genererede dokumentation kræver desuden en større process "Find fejlen -> find kodefilen i biblioteket -> find fejlen i kodefilen -> ret fejlen -> gem rettelsen -> byg dokumentationen". Det ville være nyttigt til dette at have et værktøj, så det var muligt at rette fejlen direkte i den genererede dokumentation og få denne rettelse ind i kodefilerne. En teknik til at mindske fejlene og samtidig gøre dokumentationen ensartet er at fastsætte et format og opbygge en samling af standard sætninger. Det valgte format skal ikke vælges udfra, hvor let det er at vedligeholde, men udfra, hvor meget det hjælper brugerne.

Der findes modultest til alle algoritmerne og beholderne. Modultestene til algoritmerne tester alle realisationer af de beholdere, der skal fungere til den givne algoritme og modultesten til beholderne tester alle realisationerne af den givne beholder. Grunden til, at algoritmerne ikke kun testes med en realisation af en beholder, er, at modultestene til algoritmerne ofte kunne finde fejl i realisationer af beholderne. Algoritmernes modultest virker dermed delvist også som en modultest af beholdernes realisationer. (Hvis en modultest for en algoritme fejler på for en realisation af en beholder, men ikke andre er fejlen sandsynligvis i realisationen og ikke algoritmen.)

Kapitel 2

Designvalg

Dette kapitel beskriver de overordnede designvalg, der er foretaget ved implementeringen af CPH collections og argumenterer for disse valg.

De designvalg, der blev foretaget i den nuværende udgave af CPH collections, kan deles op i to primære kategorier, disse er struktur- og ydelsevalg (*performance*).

Strukturvalg handler om, hvordan bibliotekets komponenter interagerer med hinanden og grupperes sammen. Der er to strukturer i CPH collections, den som ses af brugerne og den som ses af udviklerne.

Målet for strukturen, der ses af brugerne, er, at den skal øge den uvidenhed (*cluelessness*)[15] en bruger kan have og stadig bruge biblioteket. Uvidenhed er et mål udfra anskuelsen om, at jo mindre en bruger skal vide for at bruge biblioteket, desto lettere er det at bruge og lære. Målet for strukturen, der ses af udviklerne er, at det skal være let at finde, tilføje og rette i bibliotekskomponenter.

Det er vigtigere at forbedre brugernes struktur til at øge uvidenhed end det er at gøre livet lettere for bibliotekets udviklere. (Der er flere brugere af et bibliotek end udviklere og brugerne er grunden til bibliotekets eksistens, hvis dette havde være et virkeligt bibliotek tiltænkt brugere.)

Strukturvalg foretaget for biblioteket er også foretaget i håbet om, at de opfylder et eller flere af principperne i den følgende liste. Listen består af råd til, hvad en god grænseflade til et bibliotek bør opfylde:

Let at lære [3]: Det skal være let for en bruger at lære at bruge biblioteket.

Let at bruge [3]: Det skal være let at bruge biblioteket.

Let at læse og vedligeholde kode[3]: Det skal være let for en bruger at vedligeholde sin kode, der bruger biblioteket.

Svært at misbruge[3]: Hvis det er svært at misbruge biblioteket, er sandsynligheden for at det sker mindre.

Let at huske [5]: Det skal være let at huske, hvordan biblioteket bruges.

Lede til læslig kode [5]: Det skal være let at læse kode, der bruger biblioteket.

Ydelsesvalg handler om optimering af bibliotekets ydelse som for eksempel køretid. Ydelsvalg handler som regel om et valg mellem forskellige teknikker at skrive den samme kode på. For at finde ud af hvilken teknik, der er den mest optimale, bliver teknikkerne testede i en ydelsestest. Resultatet af testene kan læses i valget.

Den sparede tid mellem to teknikker i testene er brøkdele af et sekund, og det er derfor bedre at vurdere effekten af et valg udfra den procentvise forskel mellem to teknikker.

Ydelsestestene tester ikke kode i CPH collections, men derimod kode skrevet på de forskellige teknikker udelukkende for ydelsestesten. Det antages at besparelsen, der findes ved testningen af teknikkoden, kan videreføres til biblioteket, hvis teknikkkoden bruges.

Et ydelsesvalg bør ikke foretages, hvis valget samtidig gør brugen af biblioteket mere besværlig.

> Do not warp API to gain performance. – [3]

2.1 Strukturvalg

I denne sektion beskrives de vigtigste strukturvalg foretaget i den nuværende udgave af CPH collections.

2.1.1 Opdeling i pakker

Komponenterne (klasser og funktioner), der udgør CPH collections, opdeles i forskellige pakker udfra deres funktionalitet. (I et bibliotek placeres bøger også i kategorier; fag- og skønlitteratur i hver sin afdeling.) Grundene til dette valg er:

- Det bør gøre det lettere for udviklere at finde eller tilføje et komponent.
- Det skaber struktur at have komponenter opdelt efter deres funktionalitet. Biblioteket bliver dermed lettere at lære og at huske.
- At dele komponenterne i et programbibliotek op i pakker er brugt i mere kendte biblioteker som for eksempel *.Net*.

Effekten af dette valg er, at biblioteket er opdelt i otte hovedpakker:

Beholdere (`container`): Indholder klasserne, der implementerer beholdertyper.

Algoritmer (`algorithm`): Indholder funktionerne, der implementerer algoritmer defineret i C++0x specifikationen [2].

Iteratorer (`iterator`): Indholder klasser, der implementerer iteratorer til beholderne i beholderpakken, men også én klasse, der implementerer iteratorfabrikken 5.6.2 (s. 93), og hjælpe funktioner til fabrikken.

Fabrikker (`factory`): Indholder klasser til fabrikker, der kan returnere instanser af klasserne i beholderpakken.

Sammenligningsfunktioner (`comparator`): Indholder en bestemt type af funktioner, der foretager en sammenligning mellem to inddataer der gives til dem og returnerer sandt eller falsk, som et resultat af sammenligningen mellem de to værdier. Funktionerne kaldes for sammenligningsfunktioner og kan bruges som argument til metoder eller funktioner, der forventer en sammenligningsfunktionalitet (*comparator*).

Verificeringsfunktioner (`verify`): Indholder verificationsfunktioner til at verificere, at en værdi er af en bestemt type (beholder).

Hjælpefunktioner (`utility`): Indholder hjælpe funkioner.

Feljhåndtering (`_error`): Indholder klasser til at håndtere sproget brugt i fejlbeskeder.

I disse hovedpakker er der underpakker for eksempel har beholderpakken en underpakke `abc`. Alle pakker der hedder noget med `abc` indholder kun abstrakte klasser.

2.1.2 Private objektvariabler

Alle variabler (*fields*) i et objekt er private, hvis en bruger skal kunne sætte eller hente værdien af en variabel skal objektet have metoder til at sætte og hente denne værdi og eventuelt Python-egenskaber (*property*) til at kalde disse metoder. Ideen til dette valg er fra [15](s. 70) og [3], se citat. Ved valget opnås det, at det bliver sværere for brugerne at misbruge biblioteket. Grundene til dette valg er:

- Hvis variabler ikke er private må det forventes, at brugere gør brug af dem og dermed bindes eksistensen af variabelnavnet. Denne binding gør, at variablens navn aldrig kan ændres i implementeringen, men også, at alle implementeringer, der kan bruges det givne sted, skal have en variabel eller egenskab med det givne navn.
- Det ikke muligt at teste om brugeren sætter en variabel til en lovlig værdi på tidspunktet den sættes, men først når variablen bruges [15].
- Det er ikke muligt at gøre det umuligt at sætte en variabel.

> Make classes and members as private as possible.
> Public classes should have no public fields. – [3]

Effekten af dette valg for implementeringer skrevet i Python er, at der skrives en bundstreg (_) foran alle variabelnavne. Det er ikke muligt i Python at gøre noget reelt privat, men det er forstået, at elementer, der starter med en bundstreg, er private. (Medmindre de starter med to bundstreger.)

2.1.3 En funktionalitet en indgang

En funktionalitet skal kun have en indgang. Grundene til dette valg er:

- Hvis en funktionalitet kan opnåes på mere end en måde, kan der også være mere end et sted som skal opdateres/rettes ved fremtidige udvidelser. Biblioteket er mere krævende at vedligeholde, hvis der er flere indgange.
- Hvis en funktionalitet kan opnåes på mere end en måde, så bliver bibliotekets grænseflade støre uden at udvide, hvad det kan. En større grænseflade er svære at lære og huske.
- Kode der bruger biblioteket kan se forskellig ud, men stadig opnå det samme, og dermed bliver koden mindre læslig.

Effekten af dette valg er, at metoder, der kaldes af en egenskab, skrives som private. (Egenskaben er den indgang, der forventes brugt af brugerne) Den eneste undtagelse for reglen om én funktionalitet én indgang er, at alle implementeringer af beholdere har to metoder til at få antallet af elementer. Den ene metode `__len__` implementerer, at `len` kan kaldes med beholderen og den anden `size` er for at ligne C++0x specifikationen [2].

2.1.4 Frie iteratorer

I CPH collections er iteratorer frie fra deres beholdere. For at få en iterator til en beholder, skal iteratortypen findes og beholderen gives til denne type. Dog implementerer alle beholdere metoden `__iter__`, der returnerer en forlænsiterator til beholderen. Dette er forskelligt fra C++0x specifikationen [2], hvor en iterator til en beholder kan fåes gennem beholderens grænseflade (Metoderne `begin`, `end`, `rbegin` og `rend`). Grundene til det valg er:

- Da grænsefladen ikke kan ændres begrænser dette brugerens muligheder til en forlæns- eller baglænsiterator. (I C++ kan en grænseflade ikke ændres, men i Python er det muligt.)
- I den oprindelige udgave af CPH collections implementerede beholderne funktionalitet til at sætte typen af iterator, der blev returneret af `__iter__`. Denne løsning blev fjernet af to grunde. Den ene grund er, at for at få en iterator til en beholder skulle brugerne finde en iteratortype, der virkede til beholderen, sætte denne i beholderen og kalde `__iter__` metoden. Den anden grund er, at algoritmer i CPH collections, der bruger samlinger af elementer, bliver stillet overfor problemet om de skal bruge den iterator, der er i samlingen (beholderen), eller om de skal sætte samlingens iterator til en anden og bruge denne.

Effekten af dette valg er, at brugere, der vil have en forlænsiterator til en beholder, blot kan kalde dennes `__iter__` metode. Dette gør det muligt at skrive kode som i listing 2.1 (s. 8) og den anden effekt ville have været, at for at få en speciel iterator til en beholder skulle brugeren først have fundet iteratortypen og givet beholderen til denne.

Listing 2.1: For-løkke

```
for element in container:
    .
    .
    .
```

2.1.5 En pakkes brugerindhold er første niveau

Det indhold, som en bruger af en pakke (*package*) må forventes at bruge, er på første niveau i pakken. Grundene til det valg er:

- Ved valget af kodestil, hvor funktioner og klasser er i, hver deres fil, ville effekten af det valg være, at en bruger for at kalde en funktion skulle importere modulet med funktionen fra pakken og kalde funktionen gennem modulet, se listing 2.2.

  Listing 2.2: Funktion i modul

  ```
  import cph_collections.algorithm.reverse
  li = [1, 2, 3]
  cph_collections.algorithm.reverse.reverse(li)
  ```

For at undgå dels, at skulle skrive en `import`-erklæring af den enkelte funktion, men i særdeleshed for at undgå, at brugere skal skrive `reverse.reverse`. Importerer `algorithm` pakken `reverse` modulet og opretter en reference kaldet `reverse` til funktionen `reverse`, se listing 2.3.

Listing 2.3: Oprettelse af reference `reverse`; fra `cph_collections/algorithm/__init__.py`

```
from cph_collections.algorithm.reverse import reverse
```

Effekten af dette valg er, at brugere kan skrive som i listing 2.4.

Listing 2.4: Funktion importeret i pakke

```
import cph_collections.algorithm
li = [1, 2, 3]
cph_collections.algorithm.reverse(li)
```

2.1.6 Skjul typer med fabrikker

For alle typer af beholdere og iteratore oprettes der fabrikker. Disse fabrikker kan returnere alle realisationer af deres type og alle realisationer af deres type kan registreres i fabrikken. Det betyder at brugerne kan abstrahere væk fra navnet på en realisation. Ideen til dette design er fra [15] (s. 107). Grundene til det valg er:

- Eftersom, at dette er et eksperimentelt programbibliotek, så vil der eksistere flere realisationer af den enkelte type. Uden fabrikker vil dette kræve, at brugerne kender og bruger navnet på disse typer, at brugerne vil skulle ændre i deres kode for at ændre realisation.
- Ved brug af fabrikker gøres det muligt at implementere et intelligent valg af realisation i fabrikken ud fra parametre som for eksempel hukommelsesbrug.
- Omkostningen ved en fabrik ligger kun i oprettelsen af en instans, der er ingen senere omkostninger for instansen.

Effekten af dette valg er, at der findes fabrikpakken og én fabrik til iteratorer i iteratorpakken. Hver ny implementering af en iterator eller beholder skal registreres i deres respektive fabrikker. Dette kræver ændringer i fabriksmodulerne for at importere modulerne med de nye implementeringer.

2.1.7 Beholderfabrikker er i første niveau af biblioteket

Beholderfabrikker, se sektion 5.6.1 (s. 90), er på første niveau af CPH collections. Grundene til det valg er:

- Beholderfabrikkerne er det, som brugerne anses at skulle bruge oftest.
- Når beholderfabrikkerne er på første niveau mindskes den viden en bruger behøver – for at bruge biblioteket – til blot biblioteket- og fabrikkens navn.

Effekten af dette valg er, at brugere kan skrive kode som i listing 2.5 i stedet for som i listing 2.6. (Fabrikpakken følger designvalget i sektion 2.1.5 (s. 8).)

Listing 2.5: Brug af første niveau fabrik.

```
import cph_collections
vector = cph_collections.Vector()
```

Listing 2.6: Fabrikkens rigtige niveau.

```
import cph_collections.factory
vector = cph_collections.factory.Vector()
```

2.1.8 Global undtagelsesstyring

Alle undtagelser instansieres i en global instans, men rejses af stedet fejlen sker. Ideen til at bruge global undtagelsesstyring er fra [15] (s. 35), hvor tekstbeskeder gemmes i en global tilgængelig resource i stedet for at blive skrevet direkte ind, hvor de bruges. Grundene til dette valg er:

- Det er lettere at garantere en ens ordlyd af undtagelser på tværs af biblioteket, når disse alle står i den samme klasse.

- Det er muligt at ændre sproget undtagelsesbeskeder skrives i. Dette er implementeret i den nuværende klasse i biblioteket, der understøtter både dansk og engelsk.

- Det er muligt at ændre undtagelser, der returneres ét sted. Dette kunne være brugbart, hvis for eksempel undtagelser skulle skiftes til bibliotek specifikke undtagelser i stedet for standard undtagelserne i Python, eller ændre indholdet af en undtagelse.

- Det er muligt at ændre, udskifte eller dekorere instansen, der er i variablen. Dette kunne bruges til for eksempel at dekorere registrering af undtagelser og eventuelt videre sending af undtagelser per email til bibliotekets udviklere.

Effekten af dette valg er, at implementeringer, der rejser en undtagelse, kalder metoder i en global instans i variabelen *error*. Metoderne returnerer undtagelser, men rejser dem ikke. Dermed er alle fejlbeskeder, biblioteket rejser, sprogligt ens når de er af den samme type af fejl. For at udvide med nye fejltyper er det nok at tilføje en metode til klassen, som *error* er en instans af. Det er muligt at ændre sproget i undtagelser ved at ændre i instansen.

2.1.9 Samlinger som argumenter til algoritmer

Algoritmer og metoder skal implementeres, så de tager en samling af elementer. Dette er i modsætning til C++0x specifikationen [2], hvor algoritmer tager iteratorer i par, og elementerne mellem de to iteratorer er samlingen. Grundene til dette valg er:

- Det sparer brugeren for at hente iteratorer til start og slut af en beholder for at bruge en algoritme.

- Det sænker antallet af argumenter en bruger skal give.

Effekten af dette valg er, at for hver par af iteratorer en funktion eller metode tager i C++0x specifikationen [2], skal den i CPH collections have ét mindre argument. Det er dermed lettere, at afvikle en funktion eller metode med en instans af beholderen. Dog accepteres iteratorer i CPH collections som samlinger, de peger på det første element i deres samling og kan itereres til det sidste. Dette betyder at funktioner eller metoder kan tage en iterator som argument, hvor de kan tage en samling. Det er ikke et problem for funktioner, som *find*, hvis forløb er fra det første til det sidste element i samlingen, men det er et problem for funktioner, som *reverse*, der bevæger sig den modsatte vej i samlingen end samlingensorden.

2.1.10 Valg af kodestil

Denne sektion beskriver kodestilen, der er valgt som standarden i CPH collections. Kodestilen, der bliver brugt, er den som beskrives i [13]. Denne stil er valgt fordi

det er den foreslåede standard for, hvordan Python-kode skrives. Dermed burde valget af kodestilen resultere i, at udviklere med erfaring indenfor Python, har lettere ved at læse bibliotekets kode og skrive kode til biblioteket. (Forudsat, at de selv bruger den valgte kodestil.)

I valget af kodestil er den pågældende stil ikke så vigtig, men det er vigtigt, at den valgte stil følges konsekvent.

> One of Guido's key insights is that code is read much more often than it is written. – [13]

Denne påstand virker fra erfaring som værende korrekt – Når kode skrives bliver den læst, men når kode rettes bliver den læst og kun dele af den rettes –. Derfor er det vigtigt, at den valgte kodestil følges konsekvent for at læsningen af koden bliver lettere. Hvis der ikke blev brugt en fast kodestil i biblioteket, så skulle læserne (udviklerne) vænne sig til en ny stil, hver gang stilen ændrede sig fra en blok af kode til en anden.

Udover reglerne i den valgte kodestil er der tilføjet flere regler. De fleste af disse regler omhandler navngivningen af argumenter. Reglerne om navngivningen er tilføjet for at opnå, at navne er selvforklarende og konsistente. Dette foreslås af [3], se citat.

> Names Should Be Largely Self-Explanatory – [3]

> Be consistent-same word means same thing – [3]

Liste af tilføjet regler:

- Funktioner og klasser skal såvidt det er muligt skrives i hver deres fil. Dette valg er for at undgå filer med mange klasser eller funktioner.
- Hvis to argumenter til en funktion eller metode er af samme type, så får det første argument navnet på typen og det andet "other", se listing 2.7. Dette betyder, at der er en konsistent navngivning af det andet argument.

  Listing 2.7: Argument navngivning; fra filen `cph_collections/algorithm/swap.py`.

```
1 def swap(collection, other):
```

- Hvis et argument til en funktion eller metode bruges til resultatet af kaldet navngives dette argument "result", se listing 2.8. Dette betyder, at der er en konsistent navngivning af resultat argumenter.

  Listing 2.8: Argument der også er resultat; fra `cph_collections/algorithm/merge.py`.

```
1 def merge(collection, other, result):
```

- Argumenter i en funktion- eller metodedefinition skal have sigende navne. Et sigende navn på et argument gør det lettere at læse koden og i Python er det muligt at binde værdier til argumentetsnavn, der hvor kaldet foretages, se listing 2.9 (s. 12). Sigende navne på argumenter er i bibliotekets tilfælde navne der indikerer, hvilket concept som en værdi givet skal opfylde. Dette betyder, at der er en konsistent navngivning af argumenter på tværs af metoder og funktioner baseret på konceptet argumenterne skal opfylde.

  Listing 2.9: Kald til en funktion, hvor værdien bindes til navnet.

```
1 def function(key):
2     pass
3
4 function(key=5)
```

- Hvis et funktionkald, funktionserklæring, metodekald eller metodeerklæring overstiger reglen om højest 80 karakterer fra [13] brydes dette ved alle kommaer og ikke kun ved nogen som vist i listing 2.10.

  Listing 2.10: Ny linie ved kommaer; fra `cph_collections/container/abc.multiset_abc.py`.

```
1     def __init__(self,
2                  collection=[],
3                  comparator=cph_collections.comparator.less,
4                  extractor=lambda x: x):
```

Erfaringer med en fast kodestil har vist, at stilen er forholdsvis let at følge med undtagelse af reglen om linier på højest 80 karakterer. Denne regel brydes tit og koden skal derefter brydes op i mindre dele. En af årsagerne til at reglen brydes hurtigt er, at indrykningen i den samme stil er fire mellemrum. (En erklæring i en for løkke i en metode har dermed allerede brugt 12 tegn på mellemrum.) En anden årsag er de lange navne, der gives elementer i biblioteket.

Til at opretholde kodestilen er det en fordel at bruge et værktøj[1], der kan teste at kodestilen holdes så dette ikke skal gøres manuelt.

2.1.11 Dokumentation

Denne sektion beskriver dokumenteringen af biblioteket. Dokumenteringen af et bibliotek er næsten ligeså vigtig som biblioteket selv. Uden dokumentation er en bruger af et bibliotek nødsaget til at gætte på indholdet af det og på, hvordan dette indhold skal bruges. Dokumentationen af et bibliotek er andet og mere end beskrivelser af dets funktioner og klasser. Det er også at beskrive bibliotekets systemkrav og, hvordan det installeres [14]. Til beskrivelserne af klasserne og funktionerne hører eksempler på hvordan de bruges. Dokumentationen kan også indholde gennemgang af, hvordan bibliotekets komponenter kan bruges i løsningen af reelle opgaver (*tutorials*) [15] (s. 37).

Denne sektion beskriver valget af dokumentationsværktøj og erfaringer med at dokumentere biblioteket.

[1] Et sådan værktøj findes til den valgte kodestil og kan hentes på siden `http://pypi.python.org/pypi/pep8`

Valg af dokumentationsværktøj

Valget af dokumentationsværktøj begrundes i denne sektion. Dette valg blev foretaget i starten af specialet. Grunden til dette er, at dokumentationen skrives til bibliotekets komponenter og i stedet for at skrive dokumentationen efter de er implementeret blev det valgt at skrive den mens de blev implementeret.

Det kunne menes, at hvornår dokumentation bliver skrevet ikke burde have effekt på, hvornår dokumentationsværktøjet bliver valgt. Dette er dog ikke korrekt da forskellige værktøjer kræver dokumentationen skrevet i forskellige formater og det er derfor lettere at vælge værktøjet først. Et skift fra et dokumentationsværktøj til et andet kan kræve, at dokumentationen bliver skrevet om til dokumentationsformatet brugt af det nye værktøj. Dermed er prisen (arbejdstid) for et skift mellem to værktøjer afhængig af mængden af den skrevne dokumentationen og forskellen i de to værktøjers dokumentationsformat. Hvis mængden er lille eller formaterne er ens er prisen lille, men hvis mængden er stor eller formatterne meget forskellige er prisen større.

Kriterierne de enkelte dokumentationsværktøjer bliver vurderet på er den online dokumentation (`HTML`-filer) der skabes, men også på, hvor let den kan modificeres, genereres og udvides. Den efterfølgende liste viser kriterierne:

Søgbar: Om den genererede dokumentation i sig selv gør en søgefunktion tilgængelig eller, om dens brugere er henvist til at bruge en ekstern søgemaskine [14].

Navigerbar: Om det er muligt fra en hvilken som helst del af dokumentationen at komme til en anden del udelukkende via navigering (klik med musen). Til dette hører dog også en vurdering af antallet af klik denne navigering tager [14].

Udvidelig: Hvor svært er det at udvide dokumentationen med nye elementer, der er her ikke nødvendigvis tale om udvidelse med dokumentation for kodeelementer.

Modificerbar: Hvor let er det at ændre udseendet af dokumentationen.

Ensartethed: Bliver dokumenteret elementer vist ens. Dokumentation skal ikke ændre udseende, da dette overrasker læseren.

Tilgængelighed: Hvor tilgængelig er værktøjet.

Dokumentationsformat: Hvilket format skrives dokumentationen i.

Let at bruge: En vurdering af, hvor svært det er at få genereret dokumentation med værktøjet.

I den efterfølgende liste vil de dokumentationsværktøjer, der blev overvejet blive beskrevet.

Pydoc: Dette dokumentationsværktøj[2] kan bruges til automatisk at generere dokumentation skrevet i kildekodefiler, men det kan også bruges direkte i Python-kode til at vise dokumentation for et kodeelement. Dette sidste er formentlig den mest udbredte brug af Pydoc.

Søgbar: Dokumentationen har ikke indbygget søgefunktionalitet, men Pydoc kan blive afviklet som en server i sig selv og så har den en søgefunktionalitet.

Navigerbar: Dokumentationen kan navigeres via klik på links. Til at navigere mellem pakker er, der ved hver visning af en pakke, en liste af links af den pågældende pakkes forældre. Til at navigere i en pakke er, der ved hver visning af en pakke, en tabel af links over pakkens direkte underpakker, moduler, funktioner og klasser. Dette betyder dog, at for at navigere til en underpakke eller funktion, kan det tage op til flere klik med musen og for at navigere mellem to forskellige steder i hierakiet skal, der først navigeres op i hierakiet og så ned.

Udvidelig: Dokumentationen kan ikke udvides med nye elementer ud over at ændre i den genererede dokumentation.

Modificerbar: Det er såvidt vides ikke muligt at ændre udseendet af dokumentationen igennem værktøjet dermed ville et sådan ønske kræve efterbearbejdelse af den genererede dokumentation.

Ensartethed: Da værktøjet bruges til automatisk at genererer dokumentation for et Python-modul (.py fil) eller Python-pakke (folder) er, der i denne automatik indbygget en hvis ensartethed. Ensartetheden rækker dog kun så langt som til dokumentationsstrengene, der er selve dokumentationen. For at dokumentation skal se ens ud skal indholdet og skrivningen af dokumentationsstrengene være ens. Det er dermed i sidste ende dokumentationsskriverens (programøren) ansvar at dokumentation altid fremstår ens.

Tilgængelighed: Pydoc kommer sammen med Python distributionen og dermed har alle, der har Python også Pydoc.

Dokumentationsformat: Formattet for værktøjet er ren tekst skrevet i dokumentationsstrenge[3].

Let at bruge: Det synes ikke let at bruge Pydoc. Dels er hjælpen til Pydoc meget lille og det, der ikke bliver beskrevet her, skal gættes eller søges andet sted. Mens værktøjet kan genere dokumentationen for en pakke, så omfatter dette ikke dokumentationen af moduler eller pakker i den, men kun, at der genereres links til modulerne og pakkernes forventede dokumentationsfiler. Dermed skal Pydoc afvikles på hver enkelt pakke og modul. (Eventuelt kunne hver dokumentationskommando skrives i et script, der når det blev afviklet ville generere dokumentationen.)

[2] Pydoc:`http://docs.python.org/library/pydoc.html`

[3] Dokumentationsstreng: En streng, der er den første erklæring i en pakke, modul, klasse eller funktion i Python, er en dokumentationsstreng.

Epydoc: Dette dokumentationsværktøj[4] kan bruges til automatisk at generere dokumentation skrevet i kildekodefiler.

Søgbar: Dokumentationen genereret med Epydoc har ikke indbygget søgefunktionalitet.

Navigerbar: Dokumentation kan navigeres via, en liste af links til alle elementer i dokumentationen der altid er tilgængelig uanset hvilken del af dokumentationen der vises.

Udvidelig: Det er delvist muligt at udvide dokumentationen da værktøjet tillader, at en enkelt hjælp side tilføjes dokumentationen.

Modificerbar: Udseendet af dokumentation kan ændres ved at give værktøjet en anden `CSS`-fil, men selve opbygningen af dokumentation kan ikke ændres udover at fjerne de før omtalte globale lister af links.

Ensartethed: Da værktøjet bruges til automatisk at generere dokumentation for et Python-modul eller Python-pakke, er der i denne automatik indbygget en hvis ensartethed. Denne ensartethed rækker dog kun så langt som til dokumentationsstrengene, der er selve dokumentationen. For at dokumentation skal se ens ud skal indholdet og skrivningen af dokumentationsstrenge være ens. Det er dermed i sidste ende dokumentationsskrivernes ansvar, at dokumentation altid fremstår ens.

Tilgængelighed: Epydoc kan hentes frit, men kommer ikke per automatik med Python.

Dokumentationsformat: Værktøjet understøtter, at dokumentationen kan skrives i forskellige formater (*text*, *epytext*, *reStructuredText*, osv) i dokumentationsstrenge.

Let at bruge: At bruge Epydoc kræver udfyldning af en konfigurationsfil (tekstfil) eller brug af en medfølgende GUI. Brug af begge dele virker forholdsvis let. Epydoc kan dog i modsætning til Pydoc automatisk dokumentere en hel Python-pakke inklusiv hele pakkens under elementer. Dette gør, at hvis biblioteket er i en hovedpakke, kan denne pakke gives i konfigurationsfilen.

Sphinx: Dette dokumentationsværktøj[5] kan bruges til at generere dokumentation. Dokumentation genereres udfra dokumentationsfiler der er separate fra kildekodefiler.

Søgbar: Dokumentation har en indbygget søgefunktionalitet.

Navigerbar: Standard opbygningen i Sphinx er sektioner og undersektioner. Navigeringen mellem disse er, at der for hver sektion kan navigeres til dennes undersektioner, den foregående sektion, den næste sektion

[4] Epydoc:`http://epydoc.sourceforge.net/`
[5] Sphinx:`http://sphinx.pocoo.org/`

og til en liste over sektioner. (Til en dybde fastsat af dokumentationsskriveren.) Det er dog muligt, at generede dokumentationssider som ikke er tilgængelige via nogen sekvens af klik fra hovedsiden, men Sphinx advarer, hvis dette er tilfældet.

Udvidelig: Eftersom dokumentation genereres fra indeholdet i dokumentationsfiler i en folder, så kan denne let udvides ved at skrive endnu en dokumentationsfil.

Modificerbar: Dokumentationen bliver genereret ved at indholdet af dokumentationen gives til et skabelonsystem (*template sytem*) der indlæser skabelonfiler. Udseendet og opbygningen af dokumentationen kan dermed ændres ved at ændre, udvide eller overskrive eksisterende skabelonfiler eller tilføje nye.

Ensartethed: Da indholdet af dokumentationen gives til et skabelonsystem vil identiske elementer altid se ens ud. Dog for at opnå fuldkommen ensartethed skal dokumentationsskriveren skrive indholdet ens.

Tilgængelighed: Sphinx kan hentes frit, men kommer ikke per automatik med Python.

Dokumentationsformat: Værktøjet understøtter, at dokumentationen kan skrives i *reStructuredText*.

Let at bruge: At bruge Sphinx kræver udfyldning af en konfigurationsfil (.py), men også, at der for hver side i dokumentationen, skrives en dokumentationsfil og, at disse manuelt linkes sammen. I en dokumentationsfil er det muligt at få Sphinx til at hente dokumentationen fra dokumentationsstrenge skrevet i et Python-modul. I standard udgaven af Sphinx understøttes kun en delmængde af *reStructuredText* i dokumentationsstrenge i kodeelementer.

Til dette projekt blev værktøjet Sphinx valgt, selvom det kræver en smule mere arbejde. Dette valg er foretaget fra de følgende grunde:

- Det er let at udvide den dokumentationen med nye sektioner. (Erfaringsmæssigt er dette en fordel da dokumentationen så kan udvides med dele, der ikke er i kodefiler. For de andre værktøjer kunne dette i princippet opnåes ved at skrive disse i kodefiler.)
- Dokumentationsformatet `reStructuredText` kan læses som normal tekst og dermed kan dokumentation læses uden værktøjet og evt med værktøjet Pydoc.
- Det forudbestemte skabelontema er det samme tema som Python--dokumentationen bruger.

Erfaringer med dokumentation skrivning

Erfaringer opnået med skrivelse af dokumentation under projektet:

- Ved brugen af Sphinx som dokumentationsværktøj fandtes det, at den mindre del af *reStructuredText* formatet, der var tilladt i dokumentationsstrenge, var en byrde og derfor bruges en tilbygning[6] til Sphinx i stedet, denne tilbygning er fra biblioteket Numpy.

- En ofte brugt begrundelse for at skrive dokumentationen i kodefiler er, at denne dokumentation dermed altid vil være korrekt for det den dokumenterer. Denne begrundelse bygger dog på antagelsen, at dem, der retter i koden også opdaterer dokumentationen samtidig. Derudover kræver det også, at programmet, der kodes i, har indbygget stavekontrol og eventuelt grammatikkontrol.

- At Sphinx ikke kan sættes til automatisk at dokumentere en hel Python-pakke (CPH collections for eksempl.) gør, at det er tænkeligt, at ikke alle elementer er i dokumentationen. Da denne opbygning kræver vedligeholdelse af to filtræer (bibliotekets og dokumentationens). Kodeelementer kan risikerer ikke at være i dokumentationen, hvis der aldrig er oprettet en dokumentationsfil til dem, eller, hvis de under en restrukturering af bibliotekets filtræ er blevet flyttet til et andet sted.

- At sikre, at dokumentation ser ens ud, sikres bedst ved fastlæggelse af et format for, hvordan de enkelte elementer dokumenteres. Dette format kan ses i bilag C.1 og er baseret på dokumentationsstandarden brugt i Numpy[7]. Grunden til at dette format blev valgt er at det indholder grupperinger af elementerne i dokumentation under overskrifter og at det pointlister argumenter til funktioner, der skulle gøre dokumentation mere overskuelig for brugeren [14].

- At sikre, at dokumentation er sprogligt ens for elementer, kan opnås ved opbyggelse af en samling af standardiserede sætninger. Disse kan ses i bilag C.2.

- Dokumentation skrevet i kodefiler bliver ofte gentaget. For eksempel er dokumentation af en metode ens for alle implementeringer af et koncept, men den skal skrives i koden for hver enkelt implementering. (Der kan være små forskelle, som for eksempel navnet på klassen.)

- Arvehierakier kan medføre, at dokumentationen er sprogligt forskellig fra metode til metode i klasser. For eksempel har alle beholdere en metode `__len__`, der returnerer antallet af elementer, denne kunne implementeres i en beholderklasse og dokumenteres med sætningen "Returnerer antallet af elementer i beholderen." og andre metoder er individuelle for en specifik

[6] `http://pypi.python.org/pypi/numpydoc/0.3.1`
[7] `https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt`

beholder, for eksempel `push_back` for vektorer, der kunne dokumenteres med sætningen "Ligger er element sidst i vektoren.". Problemet opstår, hvis den viste dokumentation for vektoren indeholder dokumentationen for alle metoderne for vektoren altså både `__len__` og `push_back`, hvis den skal vil vektoren i dokumentationen blive kaldt skiftevis en beholder og en vektor. (I CPH collections findes beholderklassen ikke netop for at undgå dette.)

- Det valgte dokumentationsformat indeholder eksempler skrevet for brugeren. En sideeffekt af at have disse eksempler i kodefilerne er, at for hver gang strukturen af biblioteket rettes skal alle eksemplerne i filerne testes igen og eventuelt rettes. En anden effekt af at have eksempler skrevet i selve kodefilen er, at der med Python-distributionen medfølger værktøjet `doctest`[8]. Dette værktøj kan bruges til at afvikle testene i kildekoden og disse test kan dermed virke som en mindre modultest (*unittest*), men dette er ikke en modultest, da eksemplerne kun viser den forventede og mest admindelige brug af komponenten.

- Desto mere detaljeret et element bliver dokumenteret desto mere fylder dokumentationen i koden. Med det valgte dokumentationsformat betyder dette ofte, at dokumentationen af for eksempel en funktion er længere end koden. Dette er ikke nødvendigvis et problem, men i Python vil disse dokumentationsstrenge også være i de kompilerede filer (`.pyc`) om dette også betyder, at de bliver læst ind af fortolkeren og dermed optager hukommelse er uvist. (Det er muligt at få dokumentationsstrengene fjernet fra den kompilerede fil med kommandoen `-OO`, der gives til fortolkeren.)

- Der er i dokumentationen sektionerne *tutorials* og *reports*. Disse sektioner blev lavet for at teste, hvor let det var at udvide dokumentationen med sektioner. Dette er meget ikke et problem i Sphinx, da sphinx netop er lavet til at skrive dokumentationen externt fra kodebasen.

  Formålet med *tutorials* sektionen er, som før nævnt at have reele eksempler på brug af biblioteket og formålet med *reports* sektionen er, at have en sektion til reporter skrevet om biblioteket. Sektionen tester endvidere at det er muligt at indkludere filer skrevet i andre formater, som for eksempel `pdf`-filen til dette special.

2.2 Ydelsesvalg

Ydelsesvalg er designvalg foretaget for at forbedre ydelsen af et bibliotek. De eneste ydelsesvalg, der er blevet foretaget for CPH collections er valg der forbedre køretiden af koden i biblioteket.

[8] Doctest:`http://docs.python.org/library/doctest.html`

2.2.1 Korteste vej

Alle implementeringer skal bruge den korteste vej til det ønsket resultat. Grundene til dette valg er:

- Det antages, at en kortere vej er hurtigere end en længere vej.

Effekten af dette valg er:

- Klasser internt ikke bruger Python-egenskaber, men derimod metoden disse kalder eller hvis det er muligt dataværdien, som metoden bruger. Om det reelt er hurtigere for en klasse at undgå brug af egenskaber internt var uvist. Det kunne tænkes, at fortolkeren optimerede den tænkte ektra omkostning ud og derfor blev dette ydelsestestet. Koden og resultatet af denne test kan ses i bilag i F.1. Listing 2.11 viser de tre teknikker for at hente en værdi og de tre teknikker for at sætte en værdi og tabellerne 2.2 (s. 21) og 2.1 (s. 20) viser et udsnit af testresultaterne. Som det ses i tabellerne tager det mindst 5 gange så lang tid at sætte eller hente en værdi gennem en egenskab (`c` og `f`) og mindst 3 gange så lang tid gennem en metode (`b` og `e`) i forhold til at sætte og hente den direkte (`a` og `d`).

- At alle frie funktioner tester om inddataet har en metode af samme navn, hvis dette er tilfældet kaldes metoden under antagelse af, at denne er hurtigere og opnår det samme resultat.

Listing 2.11: Testede kortestevejteknikker.

```
class A(object):
    def __init__(self):
        self._i

    def _set_i(self, i):
        self._i = i

    def _get_i(self):
        return i

    i = property(_get_i, _set_i)

    def a(self, n):
        for i in xrange(0, n):
            self._i = i

    def b(self, n):
        for i in xrange(0, n):
            self._set_i(i)

    def c(self, n):
        for i in xrange(0, n):
            self.i = i

    def d(self, n):
        for i in xrange(0, n):
            p = self._i

    def e(self, n):
        for i in xrange(0, n):
            p = self._get_i()
```

```
    def f(self, n):
        for i in xrange(0, n):
            p = self.i
```

2.2.2 Hurtigste iteration

I implementeringer, hvor der itereres et fast antal gange bruges koden vist i listing 2.12.

Listing 2.12: Itererer n gange

```
1 for i in xrange(0, n):
2     #CODE
```

Grundene til valget er:

- `xrange` i modsætning til `range` opretter ikke først en liste af tal [12].
- Iterationsteknikken blev fundet at være den hurtigste i en benchmarktest. Koden og resultatet af denne test kan ses i bilag F.2 (s. 171).

Listing 2.13: Testede iterationsteknikker

```
1  def a(self, n):
2      i = 0
3      while i < n:
4          #code
5          i += 1
6
7  def b(self, n):
8      i = n
9      while i:
10         #code
11         i -= 1
12
13 def c(self, n):
14     for i in range(n):
15         #code
16
17 def d(self, n):
18     for i in xrange(n):
19         #code
```

	Antal iterationer		
Teknik	50100	500100	950100
Direkt(a)	$0,0042s$	$0,0420s$	$0,0764s$
Metode(b)	$0,0134s$ (319%)	$0,1380s$ (329%)	$0,2471s$ (324%)
Egenskab(c)	$0,0210s$ (500%)	$0,2088s$ (497%)	$0,3764s$ (493%)

Tabel 2.1: Tiden det tager at sætte en værdi i et antal iterationer for hver teknik. Procent tallet er afvigelsen fra bedste teknik (`a`).

	Antal iterationer		
Teknik	50100	500100	950100
Direkt(d)	$0,0035s$	$0,0356s$	$0,0646s$
Metode(e)	$0,0121s$ (346%)	$0,1120s$ (315%)	$0,2171s$ (336%)
Egenskab(f)	$0,0162s$ (463%)	$0,1645s$ (462%)	$0,3025s$ (468%)

Tabel 2.2: Tiden det tager at hente en værdi i et antal iterationer for hver teknik. Procent tallet er afvigelsen fra bedste teknik (`d`).

Tabel 2.3 viser et udsnit af test resultaterne, som det er vist i tabellen er teknik `d` hurtigere end de andre teknikker vist i listing 2.13 (s. 20). Effekten af dette valg er ikke særligt stort, da de fleste algoritmer i CPH collections er baseret på gennemløb over alle elementerne i en beholder og i dette tilfældet bruges Pythons iterator koncept.

2.2.3 Lokale metode- og funktionsreferencer

Når metoder eller funktioner kaldes i en løkke oprettes der en lokal reference til disse, og denne reference bruges da til at kalde metoden eller funktionen. Grunden til det valg er:

- Det er hurtigere at kalde en lokal reference end en global funktion fordi Python er hurtigere til at finde en lokalt defineret reference end en global funktion og fordi Python gør dette for hver iteration [12]. Der blev foretaget en ydelsestest af denne påstand. Koden og resultatet af testen kan ses ses i bilag F.3 (s. 173).
- Det er hurtigere at kalde en lokal reference end en metode fordi Python er hurtigere til at finde en lokalt defineret reference end en metode, og fordi Python gør dette for hver iteration [12]. Der blev foretaget en ydelsestest af denne påstand. Koden og resultatet af testen kan ses i bilag F.4 (s. 175).

Den følgende kode viser den konceptuelle kode i de to test, hvor testdelen af koden er fjernet. Metoderne a og b dækker den første grund og metoderne c, d, e og f den anden grund.

	Antal iterationer		
Teknik	50100	500100	950100
while 0 til n(a)	$0,0095s$ (148%)	$0,0939s$ (147%)	$0,1692s$ (138%)
while n til 0(b)	$0,0081s$ (127%)	$0,0807s$ (126%)	$0,1446s$ (118%)
range(c)	$0,0072s$ (13%)	$0,0757s$ (18%)	$0,1364s$ (11%)
xrange(d)	$0,0064s$	$0,0640s$	$0,1230s$

Tabel 2.3: Tiden det tager at foretage et antal iterationer for hver teknik. Procenttallet er afvigelsen fra bedste teknik (`d`).

Listing 2.14: Metode- og funktionskald referencer på forskellige niveauer

```
def fun(self, i):
    return i * 2

class A:
    def mfun(self, n):
        return i * 2

            def a(self, n):
                s = 0
                for i in xrange(n)
                    s += fun(i)
                return s

            def b(self, n):
                s = 0
                f = fun
                for i in xrange(n)
                    s += f(i)
                return s

            def c(self, n):
                s = 0
                for i in xrange(n)
                    s += self.mfun(i)
                return s

            def d(self, n):
                s = 0
                for i in xrange(n)
                    s += A.mfun(self, i)
                return s

            def e(self, n):
                s = 0
                f = self.mfun
                for i in xrange(n)
                    s += f(i)
                return s

            def f(self, n):
                s = 0
                f = A.mfun
                for i in xrange(n)
                    s += f(i)
                return s
```

I tabel 2.4 (s. 23) vises et udsnit af testresultaterne for kald til en funktionsreference. Som det ses i listing 2.14 er der to teknikker. Teknik `a`, hvor funktionens navn kaldes, og teknik `b`, hvor der oprettes en reference til funktionen og denne kaldes. Udsnittet af testresultatet viser at teknik `b` er hurtigere end teknik `a`.

I tabel 2.5 (s. 23) viser et udsnit af testresultaterne for forskellige teknikker til at kalde en metode på. I listing 2.14 ses de fire teknikker; teknik `c` kalder metoden gennem en objektvariabel, teknik `d` kalder metoden gennem klassens navn med objektet selv som første argument, teknik `e` opretter en reference til metoden gennem objektvariablen og kalder denne metode og teknik `f` opretter en reference til metoden gennem objektets klasse og kalder denne. Som det ses af udsnittet af testresultatet er `e` bedre end de andre teknikker `c`, `d` og `f`. Endvidere kan det udfra tabel 2.5 (s. 23) antages, at det er bedre at kalde en metode på et objekt (`c`) end at kalde den på klassen med objektet (`d`).

	Antal iterationer		
Teknik	50100	500100	950100
a	$0,0112s$ (111%)	$0,1666s$ (113%)	$0,3052s$ (107%)
b	$0,0101s$	$0,1481s$	$0,2861s$

Tabel 2.4: Tiden det tager at foretage et antal iterationer ved brug af lokal og global funktion reference. Procenttallet er afvigelsen fra bedste teknik (b).

	Antal iterationer		
Teknik	50100	500100	950100
c	$0,0138s$ (133%)	$0,1900s$ (130%)	$0,3670s$ (130%)
d	$0,0228s$ (228%)	$0,2853s$ (195%)	$0,5474s$ (195%)
e	$0,0104s$	$0,1466s$	$0,2822s$
f	$0,0173s$ (166%)	$0,2240s$ (153%)	$0,4323s$ (153%)

Tabel 2.5: Tiden det tager at foretage et antal iterationer ved brug af forskellige metode referencer. Procenttallet er afvigelsen fra bedste teknik (e).

2.2.4 Ingen indeks tjek

Implementeringer (vektor), der kan indekseres, og som samtidig tillader, at brugeren kan finde ud af, hvad det lovlige interval for indeks er, skal ikke foretage validering af indekset, når det bruges.

Grundene til valget er:

- I sagens natur må det forventes, at brugeren enten ved at et indeks er lovligt eller, at brugeren tester dette.
- Det koster at foretage test af indeks, og hvis det allerede er testet fra brugerens side, er det en irrelevant test. Der blev foretaget en ydelsestest af dette. Koden og resultatet af testen kan ses ses i bilag F.7 (s. 181).

Koden i listing 2.15 viser den konceptuelle kode i testen, hvor testdelen af koden er fjernet. Beholder A tester ikke indekset, beholder B tester om indekset er for højt og beholder C om det er for lavt eller for højt. Elementerne i alle tre beholdere gemmes i en Python liste, der selv tester indekset, det antages at prisen for denne test er ens for beholderne.

Listing 2.15: Beholder med og uden indeks tjek

```
1  class base(object):
2      def __init__(self, n):
3          self._elements = [e for e in n]
4
5      def __len__(self):
6          return len(self._elements)
7
8  class base(object):
9      def __init__(self, n):
10         self._elements = [e for e in n]
11
12     def __len__(self):
```

```
13          return len(self._elements)
14
15  class A(base):
16      def __getitem__(self, n):
17          return self._elements[n]
18
19  class B(base):
20      def __getitem__(self, n):
21          if n < len(self._elements):
22              return self._elements[n]
23          raise Exception
24
25
26  class C(base):
27      def __getitem__(self, n):
28          if n > -1 and n < len(self._elements):
29              return self._elements[n]
30          raise Exception
```

I tabel 2.6 vises et udsnit af testresultaterne for gennemløb af beholderne. Som det ses af testresultatet er beholder `A` hurtigere end beholder `B` med over 30% og hurtigere end beholder `C` med over 40%.

	Antal iterationer		
Teknik	50100	500100	950100
`A`	$0,0188s$	$0,1906s$	$0,3547s$
`B`	$0,0256s$ (136%)	$0,2497s$ (131%)	$0,4753s$ (134%)
`C`	$0,0271s$ (144%)	$0,2717s$ (142%)	$0,5145s$ (145%)

Tabel 2.6: Tiden det tager at foretage et gennemløb af beholdere med og uden indeks testning. Procenttallet er afvigelsen fra bedste beholder (`A`).

At fjerne indekstestningen sparer indlysende nok mest tid, når indeks tilgangen til en beholder bliver udført igen og igen, som det bliver i testen. Hvis indekset kun blev brugt en eller få gange er tiden der spares lav. Valget bygger endvidere på antagelsen om at brugere hellere vil beskytte sig med `if`-blokke, der validerer indekset, end med `try-except`-blokke, der ikke validerer indekset, men som fanger fejlen.

Kapitel 3

Beskrivelse af koncepter for abtrakte datatyper

Dette kapitel beskriver koncepterne for datatyperne, der er tilgængelige for brugerne af CPH collections. Et koncept beskriver en datatypes overordnede egenskaber, men er ikke en beskrivelse af en reel implementering af datatypen. En konceptbeskrivelse indeholder heller ikke en definering, beskrivelse eller visualisering af grænsefladen (*interface*) for en datatype, men dette vil blive gjort i de efterfølgende beskrivelser med `UML`-diagrammer over grænsefladen.

De beskrevet koncepter vil hovedsagligt være beholdertyperne fra C++0x specifikationen [2], som er i CPH collections. I disse beskrivelser vil der blive nævnt afvigelser mellem de to koncepter. I beskrivelserne af beholderkoncepterne vil der blive givet forslag til, hvordan den enkelte beholder kan implementeres.

3.1 Mixed

Mixed-konceptet dækker over, at den forventede type kan være en hvilken som helst type. (Dog begrænset til den type, der kan forventes i en given kontekst. For eksempel må det forventes, at typen af elementer, der hentes ud af en beholder (*container*), er af samme type som dem, der er lagt ind i beholderen.)

3.2 Samling

Samlingskonceptet (*collection*) beskæftiger sig med samlinger af elementer.

Samlingskonceptets egenskaber er:

Iteration: Det er muligt at iterere fra det første element i en samling til det sidste element.

For at kunne iterere fra det første til det sidste element i en samling må samlingen have en orden. Denne orden er individuel for hver type af samling.

3.3 Iterator

Iterator-konceptet dækker over at have en peger til et element i en samling og at kunne rykke til det næste eller foregående element. (Eventuelt andre afhængig af iteratortypen).

Iteratorkonceptets egenskaber er:

Næste: Det er muligt rykke iteratoren til det næste element og få dette næste element.

Invaliditet En iterator kan blive invalid. For eksempel, hvis elementet iteratoren er ved ændres, hvis iteratoren ændrer beholder eller, hvis iteratoren peger på elementer, der ikke findes.

For de efterfølgende beholdertyper gælder det, hvis beholderen er baseret på noder, at iteratoren kan blive invalid under følgende betingelser:

- Hvis den node, den er ved fjernes, fra beholderen. Eftersom iteratoren stadig peger på noden er det nødvendigt at fjerne pegere fra noden til noderne i beholderen ellers vil iteratoren kunne itereres tilbage til noderne i beholderen. I CPH collections vil dette løses ved at sætte pegerne i noden til `None`, og iteratorer, der forsøger at iterere i noder ikke i en beholder skal rejse en undtagelse. Iteratorer der peger på noder ikke i en beholder og som kan dereferenceres burde også rejse undtagelsen når de blev dereferenceret, men dette ville kræve et ekstra data felt i noden eller at beholderen fortalte iteratoren, at den var invalid. Denne type af invalid iterator kaldes i [9] for en *dangling iterator*.

- Hvis elementerne i to beholdere ombyttes. Iteratoren vil stadig pege på den samme node og itereringen vil være den samme. Derfor er iteratoren kun invalid, hvis det forventes, at iteratoren er over elementerne i beholder.

- Hvis eksterne algoritmer ved brug af iteratorer omrokkerer på ordenen i beholderen. Iteratoren vil pege på den samme node, men både elementet i noden og den forsatte iterering kan have ændret sig. Denne type af invalid iterator kaldes i [9] for en *inconsistent iterator*.

For vektorkonceptet kan en iterator implementeres ved brug af konceptets direkte indeksering. En sådan iterator vil bliver invalid under følgende betingelser:

- Ved fjernelse af elementer fra vektoren. Vektoren tillader kun at elementer fjernes fra slutningen af beholderen, så en invalid iterator vil pege på et ikke existerende indeks. I CPH collections vil iteratorer, der forsøger at itererer i ikke existesterende indeks rejse en undtagelse. Denne type af invalid iterator kaldes i [9] for en *dangling iterator*.

- Ved ombytning af elementerne i to vektorer. Iteratoren vil stadig pege på det samme indeks i den samme vektor, men dette indeks vil enten pege på

et nyt element eller et ikke existerende indeks, og den forsatte iterering vil være forskellig fra den oprindelige. Denne type af invalid iterator kaldes i [9] for en *inconsistent iterator*.

- Hvis eksterne algoritmer ved brug af iteratorer omrokkerer på ordenen i beholderen. Iteratoren vil pege på den samme node, men både elementet i noden og den forsatte iterering kan have ændret sig. Denne type af invalid iterator kaldes i [9] for en *inconsistent iterator*.

3.4 Beholder

Beholderkonceptet beskæftiger sig med de egenskaber en beholder har. Konceptet er en udvidelse af samlingskonceptet.

Udover egenskaberne fra samlingskonceptet er beholderkonceptets egenskaber:

Instantiering Det er muligt at oprette en beholder uden nogle argumenter eller med en samling som argument. Elementerne i samlingen bliver elementer i beholderen.

Sammenligning Det er muligt at foretage alle sammenligningsoperationer af to beholdere af samme specifikke koncept. Det kræver dog, for at få et meningsfuldt svar, at elementerne i beholderne indbyrdes kan sammenlignes.

Ombytning Det er muligt at ombytte elementerne i to beholdere af samme specifikke koncept og af samme implementering.

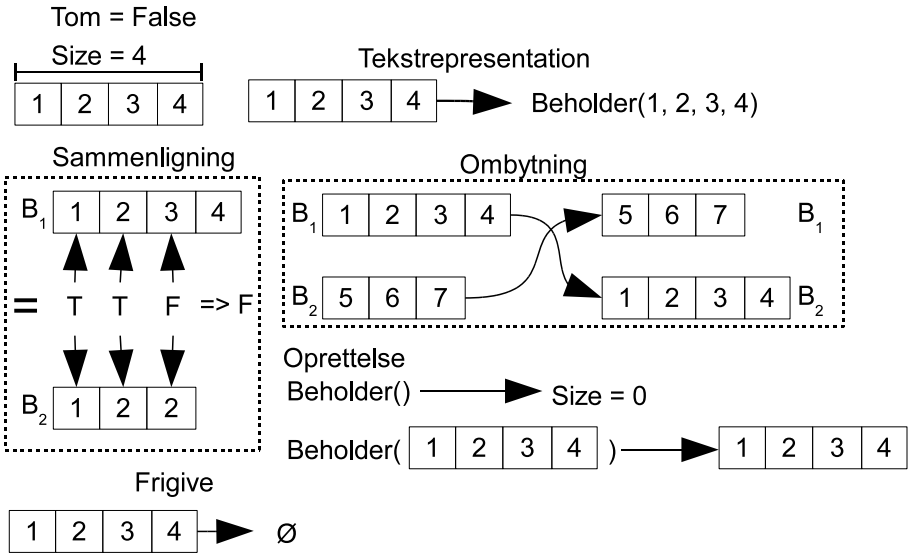
Tom Det er muligt, at få at vide om en beholder er eller ikke er tom og at tømme den.

Frigive Det er muligt at frigive en beholder og alle dens elementer.

Tekstrepræsentation Det er muligt at få en tekstrepræsentation af en beholder. Repræsentationen kan inkludere repræsentationen af et eller flere af elementerne i beholderen.

Størrelse Det er muligt få størrelsen af en beholder. (Antallet af elementer i beholderen.)

Figur 3.1 (s. 28) viser de før omtalte egenskaber for konceptet.



Figur 3.1: Beholderkonceptets egenskaber; egenskaber, der omhandler to beholdere, er omkranset af stipillede firkanter og B_1 og B_2 indikerer de to beholdere.

3.5 Vektor

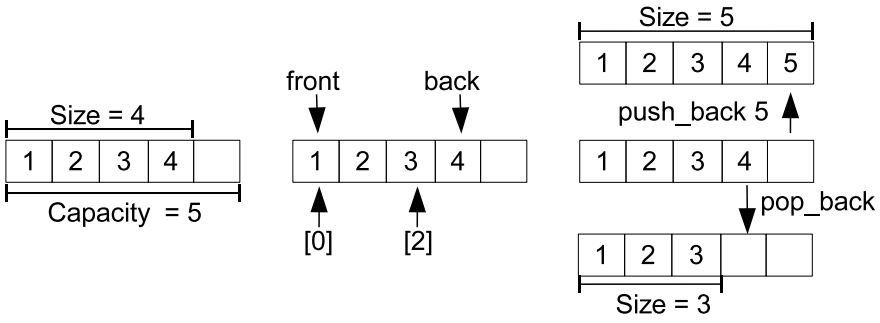
Vektorkonceptet (*vector*) er hentet fra C++0x specifikationen [2] af det samme koncept, dog er følgende egenskaber fjernet fra det oprindelige koncept; Indsætning og fjernelse af elementer på et tilfældigt sted og i starten af vektoren, disse er fjernet da de i de fleste implementeringer vil forårsage to ting, at minimum alle elementer efter de fjernede elementer skal rykkes fremad, og at iteratorer kan blive invalide. (Hvis iteratoren er baseret på indeksering vil de pege på et andet element end det de oprindeligt pegede på, hvis de i deres iterering var ved eller efter de fjernede elementer.) Metoderne i dette koncept er beskrevet i bilag B.1 (s. 122).

Vektorkonceptet er en udvidelse af beholderkonceptet og har de følgende egenskaber udover dem i beholderkonceptet:

Front Det er muligt at få og sætte det første element i vektoren.

Ende Det er muligt at få og sætte det sidste element i vektoren, men også at fjerne eller indsætte et element sidst i vektoren. Forskellen på at sætte det sidste element og indsætte et element sidst i vektoren er, at det første ændrer et eksisterende element mens det andet indsætter et element efter det sidste element.

Kapacitet Det er muligt at få og sætte kapaciteten, dette er antallet af elementer vektoren kan have. Udover muligheden for manuelt at styre kapaciteten



Figur 3.2: Vektorkonceptets egenskaber.

gør vektoren det også automatisk efter behov baseret på den pågældende implementerings udvidelses- og formindskelsespolitik.

Direkte indekserbar Det er muligt at få elementet på et bestemt indeks. Indeksene starter ved 0 for det første element i vektoren og det sidste indeks er $n - 1$, hvor n er antallet af elementer i vektoren. Eftersom elementer kun kan indsættes sidst i vektorkonceptet, er et elements indeks i vektoren konstant. (Medmindre det eksternt bliver rykket rundt.)

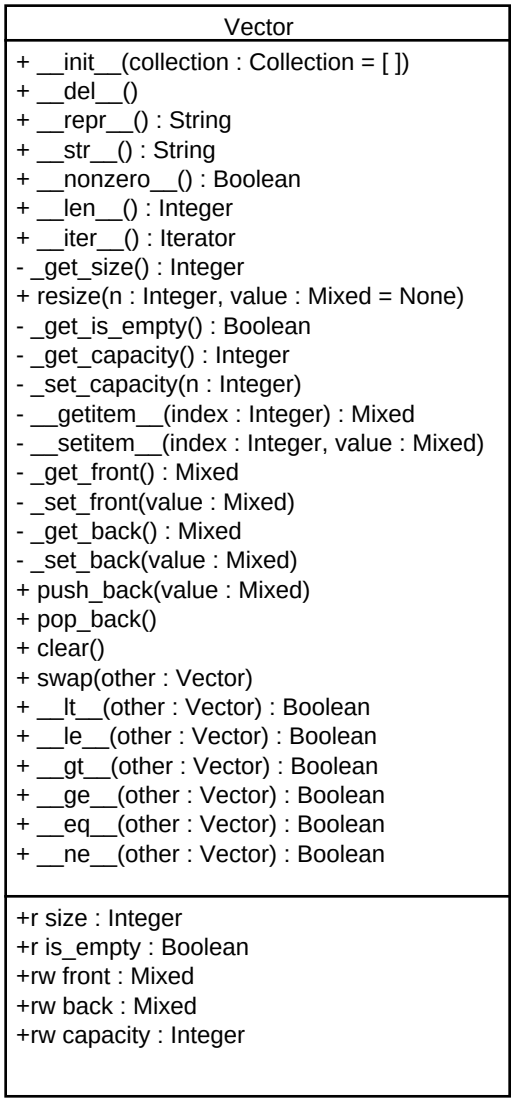
Iterering Det er muligt at iterere fra det sidste element til det første element i vektoren.

Ordenen i konceptet er, at elementet i indeks 0 er det første efterfulgt af elementet i indeks 1 osv. indtil indeks $n - 1$, hvor n er antallet af elementer i vektoren.

Egenskaben størrelse er altid positiv eller nul og altid mindre end eller lig med egenskaben kapacitet.

Figur 3.3 (s. 30) viser `UML`-diagrammet for en klasse af konceptet og figur 3.2 de før omtalte egenskaber for koceptet.

En mulig implementering af dette koncept er en kontinuerlig blok i hukommelsen, hvor elementer lægges i fra start til slut. Når blokken bliver for lille eller for stor allokeres en ny blok af passende størrelse og alle elementerne rykkes til starten af denne.



Figur 3.3: UML-diagrammet for vektorkonceptet.

3.6 Liste

Listekonceptet (*list*) er fra C++0x specifikationen [2] af det samme koncept. Metoderne i dette koncept er beskrevet i bilag B.2 (s. 125).

Listekonceptet er en udvidelse af beholderkonceptet og har de følgende egenskaber udover dem i beholderkonceptet:

Front Det er muligt at få og sætte det første element i listen, men også at fjerne eller indsætte et element i fronten af listen.

Ende Det er muligt at få og sætte det sidste element i listen, men også at fjerne eller indsætte et element i enden af listen.

Indsætning Det er muligt at indsætte et eller flere elementer på et tilfældigt sted i listen. Specielt er det muligt at kæde (*splice*) to lister af samme implementering sammen.

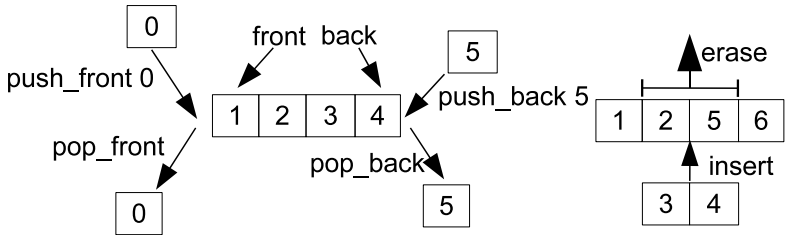
Sletning Det er muligt at slette et eller flere elementer fra et tilfældigt sted i listen.

Iterering Det er muligt at iterere fra det sidste element til det første element i listen.

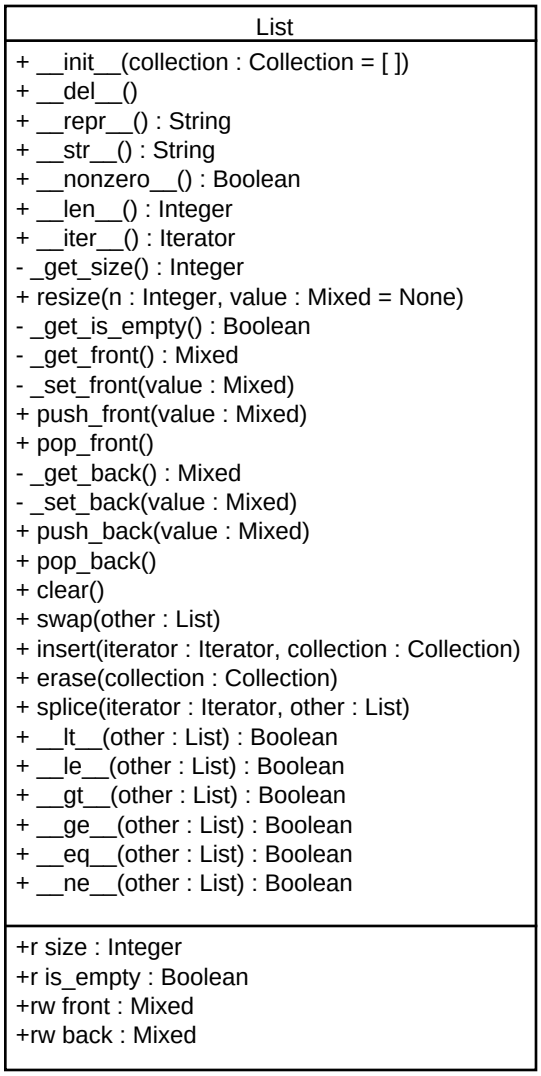
Ordenen i konceptet er, at elementer der indsættes med ende egenskaben er efter alle andre elementer og elementer der indsættes med front egenskaben er før alle andre elementer. Elementer der indsættes ved indsætning er efter det element, der gives ved indsætningen.

Figur 3.5 (s. 32) viser `UML`-diagrammet for en klasse af listekonceptet og figur 3.4 de før omtalte egenskaber for konceptet.

En mulig implementering af dette koncept er en dobbeltlinket ring.



Figur 3.4: Listekonceptets egenskaber.



Figur 3.5: UML-diagrammet for listekonceptet.

3.7 Toendetkø

Toendetkøkonceptet (*deque*) er fra C++0x specifikationen [2] af det samme koncept. Dog er følgende egenskaber fra det oprindelig koncept fjernet. Det er ikke muligt at indsætte eller fjerne elementer på tilfældige steder, og det er heller ikke muligt at bruge direkte indeksering. Metoderne i dette koncept er beskrevet i bilag B.3 (s. 127).

Toendetkøkonceptet er en udvidelse af beholderkonceptet og har følgende egenskaber udover dem i beholderkonceptet:

Front Det er muligt at få og sætte det første element i køen, men også at fjerne og indsætte et elemtent i fronten.

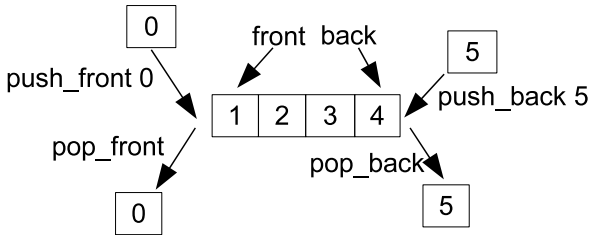
Ende Det er muligt at få og sætte det sidste element i køen, men også at fjerne og indsætte et elemtent i enden.

Iterering Det er muligt at iterere fra det sidste element til det første.

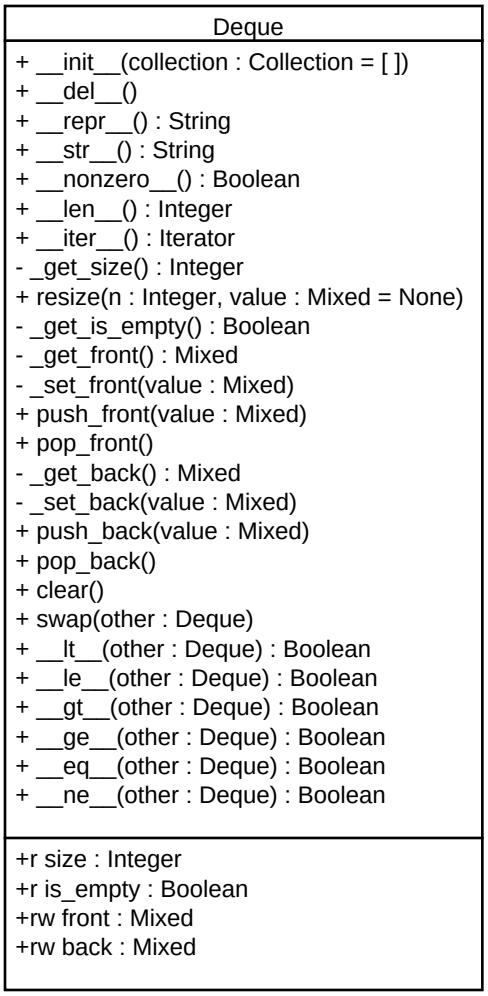
Ordenen i konceptet er, at elementer, der indsættes med ende egenskaben, er efter alle andre elementer og elementer, der indsættes med front egenskaben, er før alle andre elementer.

Figur 3.7 (s. 34) viser `UML`-diagrammet for en klasse af konceptet og figur 3.6 de før omtalte egenskaber for konceptet.

En mulig implementering af dette koncept er som en adapter af en implentering af listekonceptet, men konceptet kan også implementeres over en dobbeltlinket ring.



Figur 3.6: Toendetkøkonceptets egenskaber.



Figur 3.7: UML-diagrammet for toendetkøkonceptet.

3.8 Stak

Stakkonceptet (*stack*) er fra C++0x specifikationen [2] af det samme koncept. Dette koncept implementerer en `LIFO`-beholder. (Sidste ind er første ud). I C++0x er en stak blot en adapter af en liste eller vektor og kan ikke itereres, men i CPH collections er stakken en selvstændig beholder og kan itereres. Metoderne i dette koncept er beskrevet i bilag B.5 (s. 132).

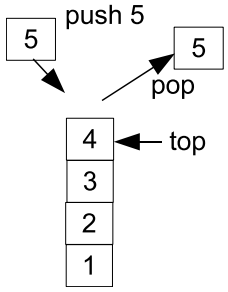
Stakkonceptet er en udvidelse af beholderkonceptet og har følgende egenskaber udover dem i beholderkonceptet:

Top Det er muligt at få og sætte det øverste element i stakken, men også at fjerne eller indsætte et element øverst i stakken.

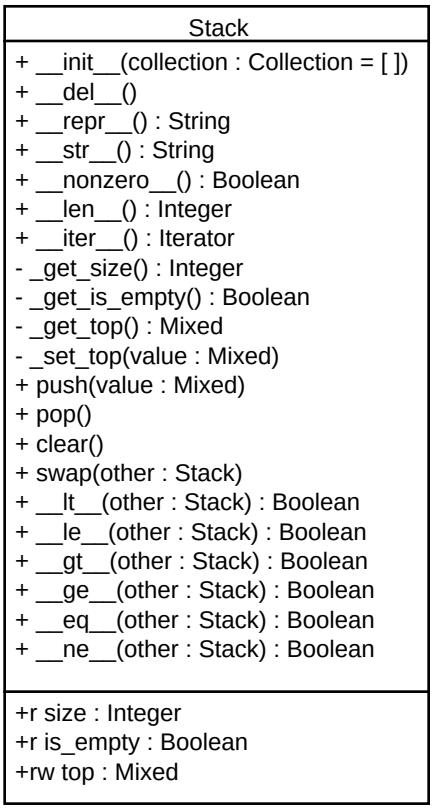
Ordenen i konceptet er, at elementer der indsættes i toppen af stakken, er før alle elementer, der er i stakken.

Figur 3.9 (s. 36) viser `UML`-diagrammet for en klasse af stakkonceptet og figur 3.8 de før omtalte egenskaber for konceptet.

En mulig implementering af dette koncept er som en adapter af en vektor eller liste, men kan også implementeres over en enkeltlinket ring.



Figur 3.8: Stakkonceptets egenskaber.



Figur 3.9: UML-diagrammet for stakkonceptet.

3.9 Kø

Køkonceptet (*queue*) er fra C++0x specifikationen [2] af det samme koncept. Dette koncept implementerer en `FIFO`-beholder. (Første ind er første ud.) I C++0x er en kø en adapter af en liste eller toendetkø og kan ikke itereres, men i CPH collections er køen en selvstændig beholder og kan itereres. Metoderne i dette koncept er beskrevet i bilag B.4 (s. 130).

Køkonceptet er en udvidelse af beholderkonceptet og har følgende egenskaber udover dem i beholderkonceptet:

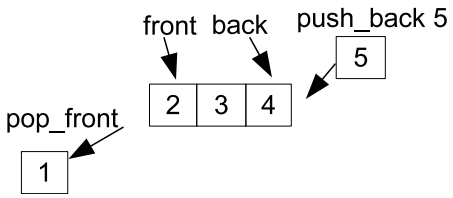
Front Det er muligt at få, sætte og fjerne det første element i køen.

Ende Det er muligt at få og sætte det sidste element i køen og at indsætte et element i enden.

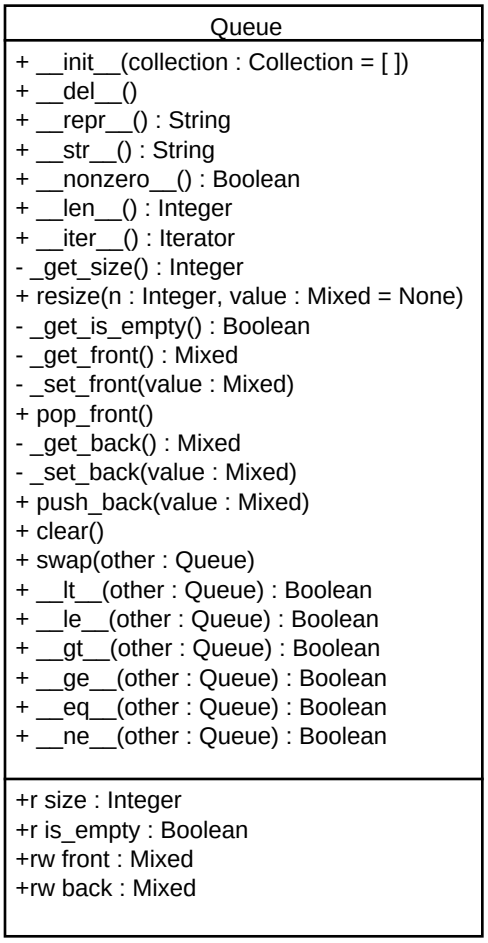
Ordenen i konceptet er, at elementer der indsættes i enden, er efter alle elementer, der er i køen.

Figur 3.11 (s. 38) viser `UML`-diagrammet for en klasse af køkonceptet og figur 3.10 de før omtalte egenskaber for konceptet.

Konceptet kan implementeres som en adapter af en liste eller toendetkø, men kan også implementeres over en enkeltlinket ring.



Figur 3.10: Køkonceptets egenskaber.



Figur 3.11: UML-diagrammet for køkonceptet.

3.10 Sæt

Sætkonceptet (*set*) er fra C++0x specifikationen [2] af det samme koncept. Dette koncept implementerer et sæt, hvor det garanteres at alle nøgler er unikke. Metoderne i dette koncept er beskrevet i bilag B.6 (s. 133). Til forskel fra C++0x specifikationen [2] er nøglen i CPH collections en del af et element i stedet for bare at være elementet. Til at hente nøglen ud af et element gives en udtræksfunktion (*extractor*) og gives denne ikke er elementet nøglen. Effekten af denne forskel er ved egenskaber, der bruger nøglerne, her kan en nøgle gives i stedet for et element, der har nøglen. (Hvis for eksempel sættet er over lande og nøglen for hvert land er dets navn, så vil egenskaber, der forventer at få en nøgle, skulle gives navnet på landet altså en tekst og ikke gives et land med det ønskede navn sat.)

Sætkonceptet er en udvidelse af beholderkonceptet og har følgende egenskaber udover dem i beholderkonceptet:

Sammenligning Det er muligt at få sammenligningsfunktionen, men denne kan kun sættes ved oprettelsen.

Udtræk Det er muligt at få udtræksfunktionen, men denne kan kun sættes ved oprettelsen.

Indsæt Det er muligt at indsætte et element.

Tælling Det er muligt at få antallet af elementer der har en given nøgle. (Eftersom at nøgler er unikke er dette højest én.)

Nedregrænse Det er muligt at få en iterator til det første element, hvis nøgle ikke er før en given nøgle i sættets orden.

Øvregrænse Det er muligt at få en iterator til det første element, hvis nøgle er efter en given nøgle i sættets orden.

Omkransing Det er muligt at få iteratorerne til elementerne, der omkranser en given nøgle. (Nedre- og øvregrænse.)

Søgning Det er muligt at få en iterator til det første element, der har en given nøgle.

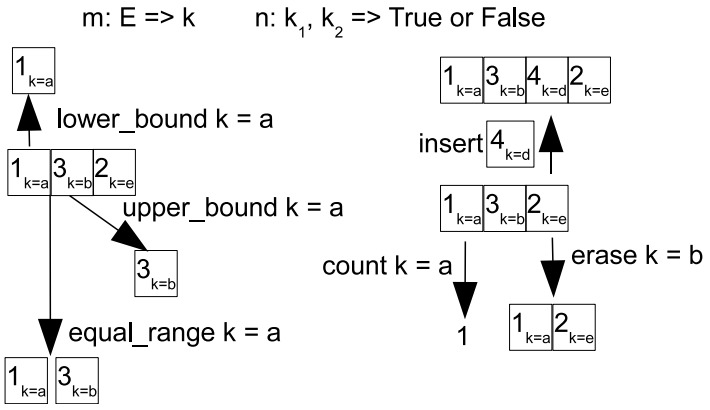
Fjerne Det er muligt at fjerne et element, der har en given nøgle.

Ordenen i konceptet er nøglernes orden. Dette vil sige, at givet to elementer så vil deres orden være den, der når deres nøgler gives til sammenligningsfunktionen, får denne til at returnere sandt.

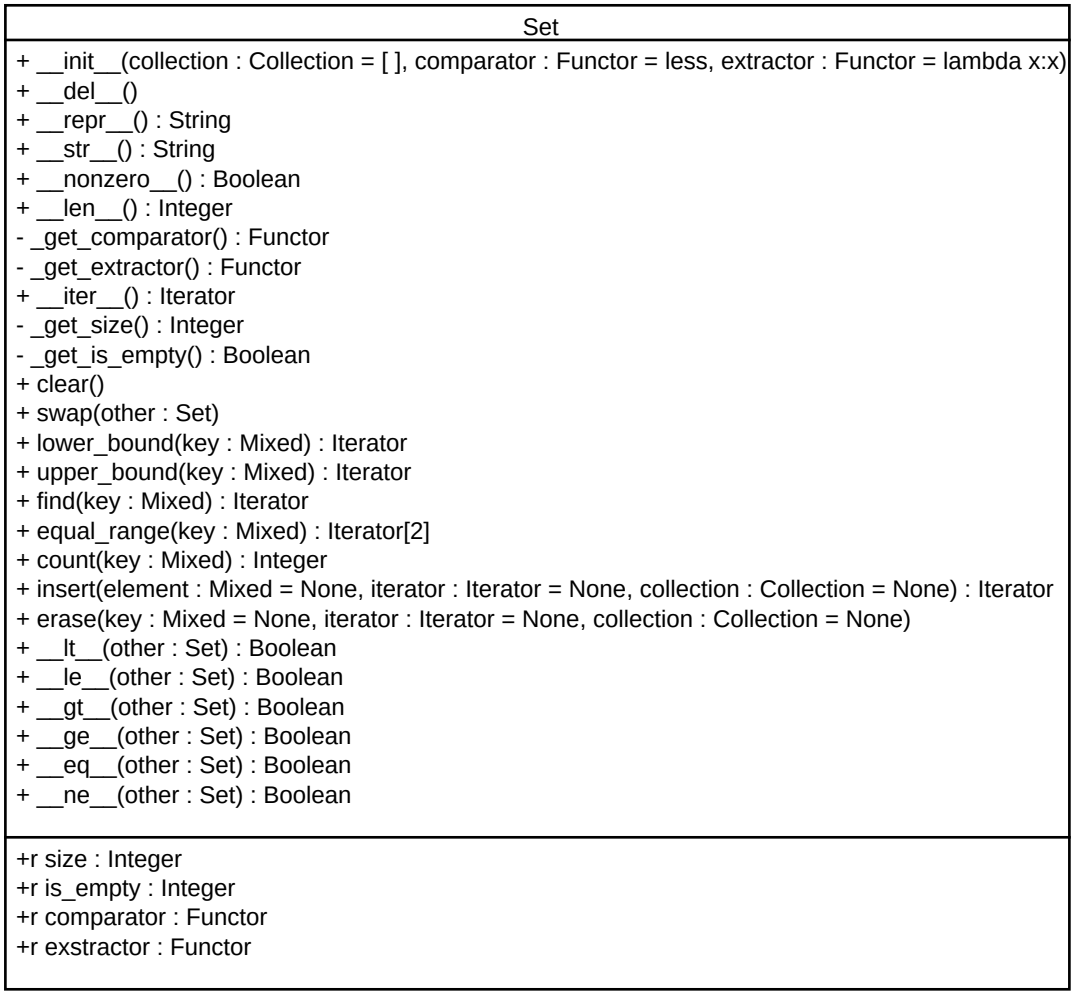
I en komplet implementering af et sæt bør det ikke være muligt at ændre i et element, når det først er lagt i sættet. Dette ville kunne ændre elementets nøgle og dermed ødelægge ordenen i sættet.

Figur 3.13 (s. 41) viser `UML`-diagrammet for en klasse af sætkonceptet og figur 3.12 (s. 40) de før omtalte egenskaber for konceptet.

Konceptet kan implementeres som en adapter af en skipliste eller et binært søgetræ.



Figur 3.12: Sætkonceptets egenskaber. Udtræksfunktionen (m) tager et element (E) og returnerer dets nøgle (k), og sammenligningsfunktionen (n) tager to nøgler (k_1 og k_2) og returnerer sandt, hvis den første nøgle er før den anden i ordenen. Egenskaber, der returnerer en iterator, er vist med elementet denne iterator peger på.



Figur 3.13: UML-diagrammet for sætkonceptet.

3.11 Multisæt

Multisætkonceptet (*multiset*) er fra C++0x specifikationen [2] af det samme koncept. Dette koncept implementerer et sæt, hvor det ikke garanteres, at alle nøgler er unikke. Metoderne i dette koncept er beskrevet i bilag B.7 (s. 136). Til forskel fra C++0x specifikationen [2] er nøglen i CPH collections en del af et element i stedet for bare elementet. Til at hente nøglen ud af et element gives en udtræksfunktion og gives denne ikke er, elementet nøglen. Effekten af denne forskel er ved egenskaber, der bruger nøglerne, her kan en nøgle gives i stedet for et element, der har nøglen. (Hvis for eksempel multisættet er over personer og nøglen for hver person er dennes alder, så vil egenskaber, der forventer at få en nøgle skulle gives alderen altså et tal og ikke gives en person med den ønskede alder sat.)

Multisætkonceptet er en udvidelse af sætkonceptet og ændrer følgende egenskaber i sætkonceptet:

Tælling Det er muligt at få antallet af elementer, der har en given nøgle.

Søgning Det er muligt at få en iterator til det første element, der har en given nøgle.

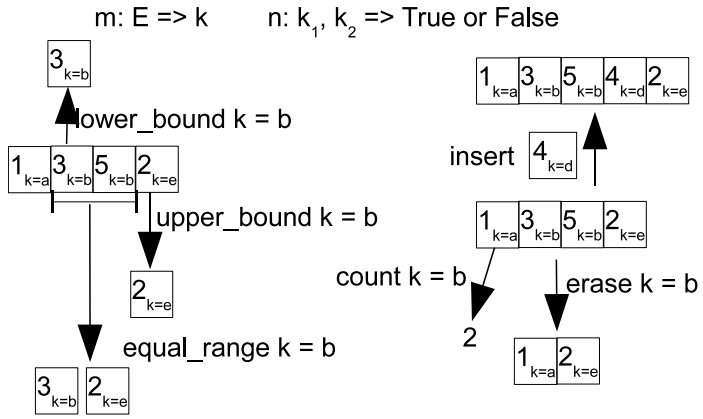
Fjerne Det er muligt at fjerne alle elementer, der har en given nøgle.

Ordenen i konceptet er, at gives sammenligningsfunktionen nøglen af to på hinanden følgende elementer, returnerer den sandt, hvis udtrækket fra det første element gives først, eller, hvis den returnerer falsk uanset rækkefølgen, så er ordenen den rækkefølge elementerne blev lagt i multisættet. Denne sidste regel definerer ordenen når nøgledelen af elementerne er ens.

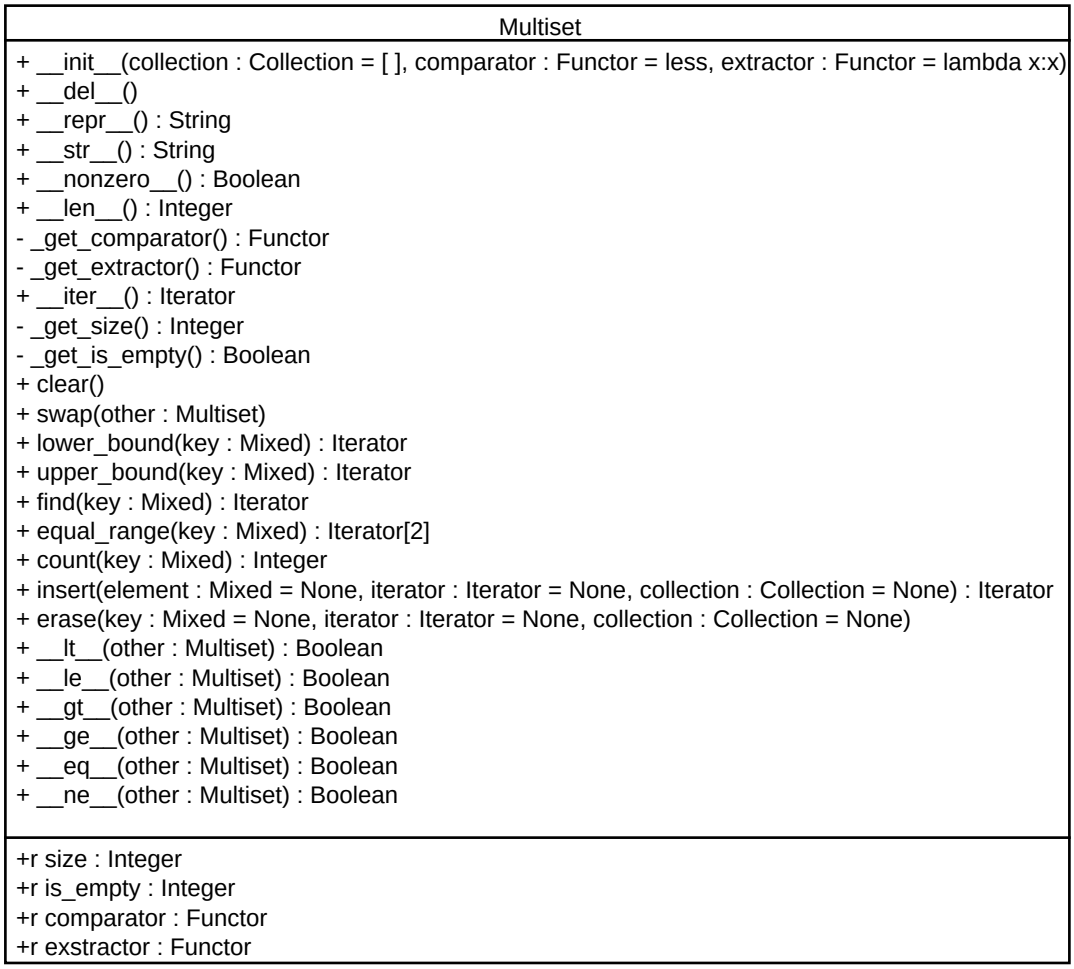
I en komplet implementering af et multisæt bør det ikke være muligt at ændre i et element, når det først er lagt i multisættet. Dette ville kunne ændre elementets nøgle og dermed ødelægge ordenen i multisættet.

Figur 3.15 (s. 44) viser `UML`-diagrammet for en klasse af multisætkonceptet og figur 3.14 (s. 43) de før omtalte egenskaber for konceptet.

Konceptet kan implementeres som en adapter af en skipliste eller et binært søgetræ.



Figur 3.14: Multisætkonceptets egenskaber. Udtræksfunktionen (m) tager et element (E) og returnerer dets nøgle (k), og sammenligningsfunktionen (n) tager to nøgler (k_1 og k_2) og returnerer sandt, hvis den første nøgle er før den anden i ordenen. Egenskaber, der returnerer en iterator, er vist med elementet denne iterator peger på.



Figur 3.15: UML-diagrammet for multisætkonceptet.

3.12 Kort

Kortkonceptet (*map*) er fra C++0x specifikationen [2] af det samme koncept. Dette koncept implementerer et kort. Et kort bliver dannet af to datatyper, der sammen danner et par. Den første datatype i dette par er nøglen til parret i kortet, og den anden datatype er parrets element. Det garanteres, at alle nøgler er unikke. Metoderne i dette koncept er beskrevet i bilag B.8 (s. 138). Til forskel fra sætkonceptet er nøglen i et kort en separat dataenhed fra elementet, og et kort kan derfor bruges, når nøglen ikke kan hentes fra elementerne, der skal gemmes. Denne forskel udmøntes også i, at der ikke er nogen udtræksfunktion i kortet, da det forventes at den datastruktur, der gives som nøgle, er nøglen.

Kortkonceptet er en udvidelse af beholderkonceptet og har følgende egenskaber udover dem i beholderkonceptet:

Sammenligning Det er muligt at få sammenligningsfunktionen, men denne kan kun sættes ved oprettelsen.

Indsæt Det er muligt at indsætte et par.

Tælling Det er muligt at få antallet af par, der har en given nøgle. (Eftersom at nøgler er unikke, er dette højest én.)

Nedregrænse Det er muligt at få en iterator til det første par, hvis nøgle ikke er før et given nøgle.

Øvregrænse Det er muligt at få en iterator til det første par, hvis nøgle er efter en given nøgle.

Omkransing Det er muligt at få iteratorerne til parrene, der omkranser en given nøgle. (Nedre- og øvregrænse.)

Søgning Det er muligt at få en iterator til parret, der har en given nøgle.

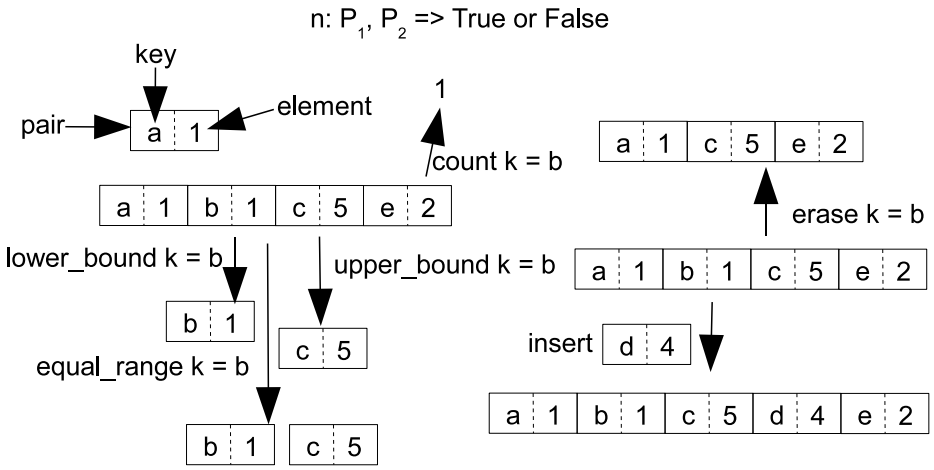
Fjerne Det er muligt at fjerne et par i sættet, der har en given nøgle.

Ordenen i konceptet er, at to på hinanden følgende par resulterer i at sammenligningsfunktionen givet parrenes nøgler returnerer sandt.

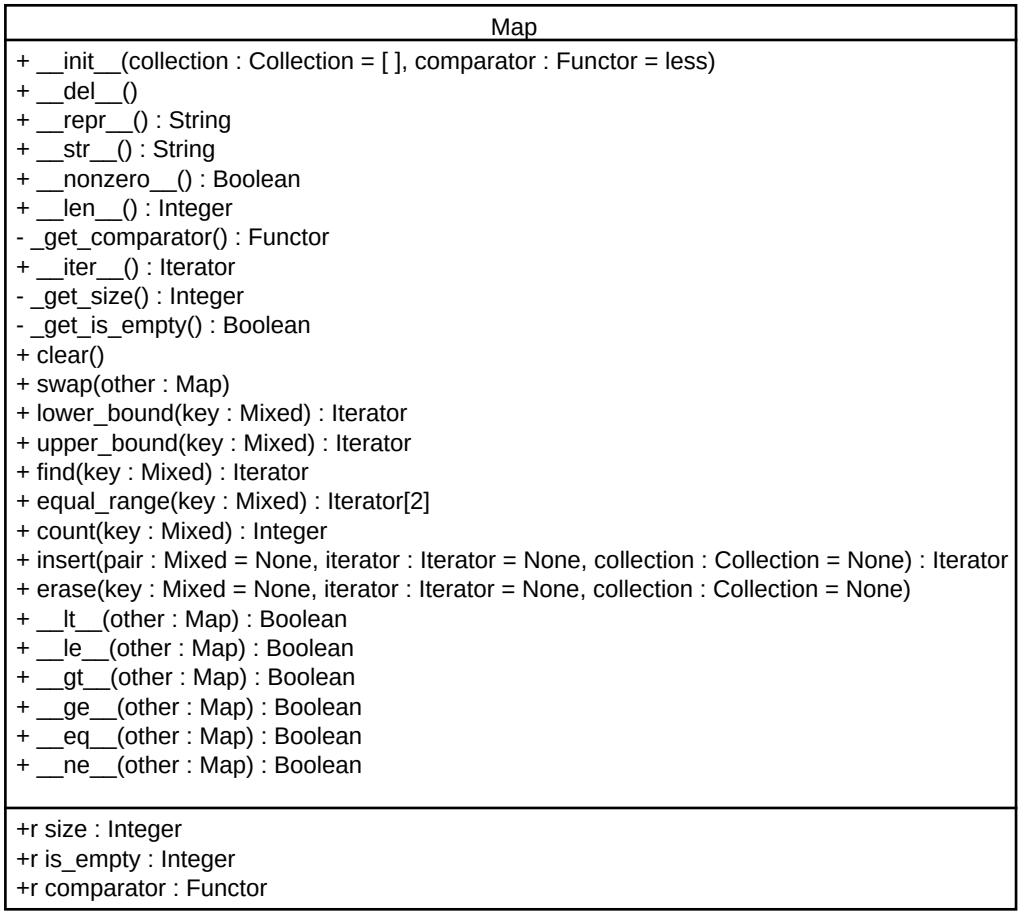
I en komplet implementering af et kort bør det ikke være muligt at ændre i nøglen i et par, men at ændre i elementet i et par er muligt, når det først er lagt i kort. Dette ville kunne ændre parrenes orden i kortet.

Figur 3.17 (s. 47) viser `UML`-diagrammet for en klasse af kortkonceptet og figur 3.16 (s. 46) de før omtalte egenskaber for konceptet.

Konceptet kan implementeres som en adapter af en skipliste eller et binært søgetræ. Det kunne menes, at konceptet kunne implementeres som en adapter til sætkonceptet, men her er det vigtigt at huske, at en korrekt sæt implementering ikke bør lade det være muligt at rette i dets elementer, og da vil det ikke være muligt at rette i elementdelen af et par fordi, at parret er et element i et sæt.



Figur 3.16: Kortkonceptets egenskaber. Sammenligningsfunktionen (n) tager nøgledelen i to par (P_1 og P_2) og sammenligner dem, hvis det første pars nøgle er før det andet i ordenen returneres sandt. Egenskaber, der returnerer en iterator, er vist med parret, denne iterator peger på.



Figur 3.17: UML-diagrammet for kortkonceptet.

3.13 Multikort

Multikortkonceptet (*multimap*) er fra C++0x specifikationen af det samme koncept. Dette koncept implementerer et multikort. Et multikort bliver dannet af to datatyper, der sammen danner et par. Den første datatype i dette par er nøglen og den anden er elementet. Metoderne i dette koncept er beskrevet i bilag B.9 (s. 141). Til forskel fra multisætkonceptet er nøglen i et multikort en separat dataenhed fra elementet, og et multikort kan derfor bruges, når nøglen ikke kan hentes fra elementerne, der skal gemmes. Denne forskel udmøntes også i, at der ikke er nogen udtræksfunktion i multikortet, da det forventes, at den datastruktur, der gives som nøgle, er nøglen.

Multikortkonceptet er en udvidelse af kortkonceptet og ændrer følgende egenskaber i kortkonceptet:

Tælling Det er muligt at få antallet af par, der har en given nøgle.

Søgning Det er muligt at få en iterator til det første par, der har en given nøgle.

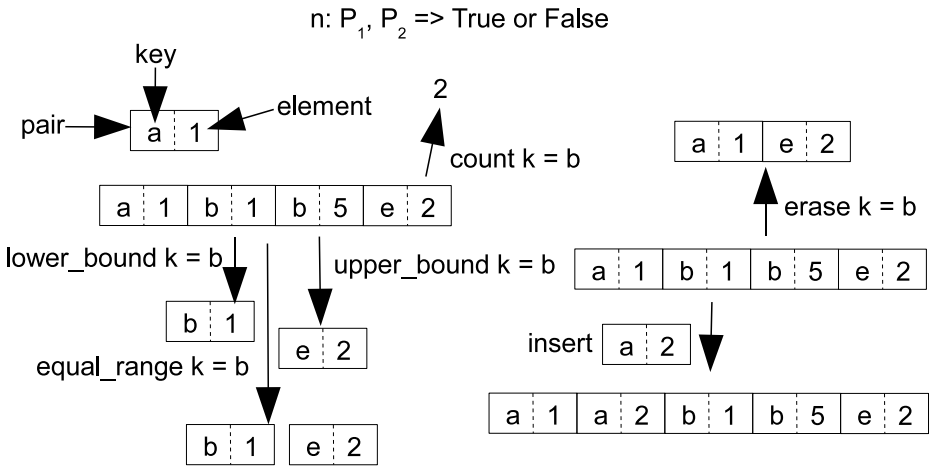
Fjerne Det er muligt at fjerne alle par, der har en given nøgle.

Ordenen i konceptet er, at gives sammenligningsfunktionen nøgledelen af to på hinanden følgende par returnerer den sandt, hvis nøgledelen fra det første par gives først, eller, hvis den returnerer falsk uanset rækkefølgen, så er ordenen den rækkefølge parrene blev lagt i multikortet. Denne sidste regel definerer ordenen, når nøgledelen af parrene er ens.

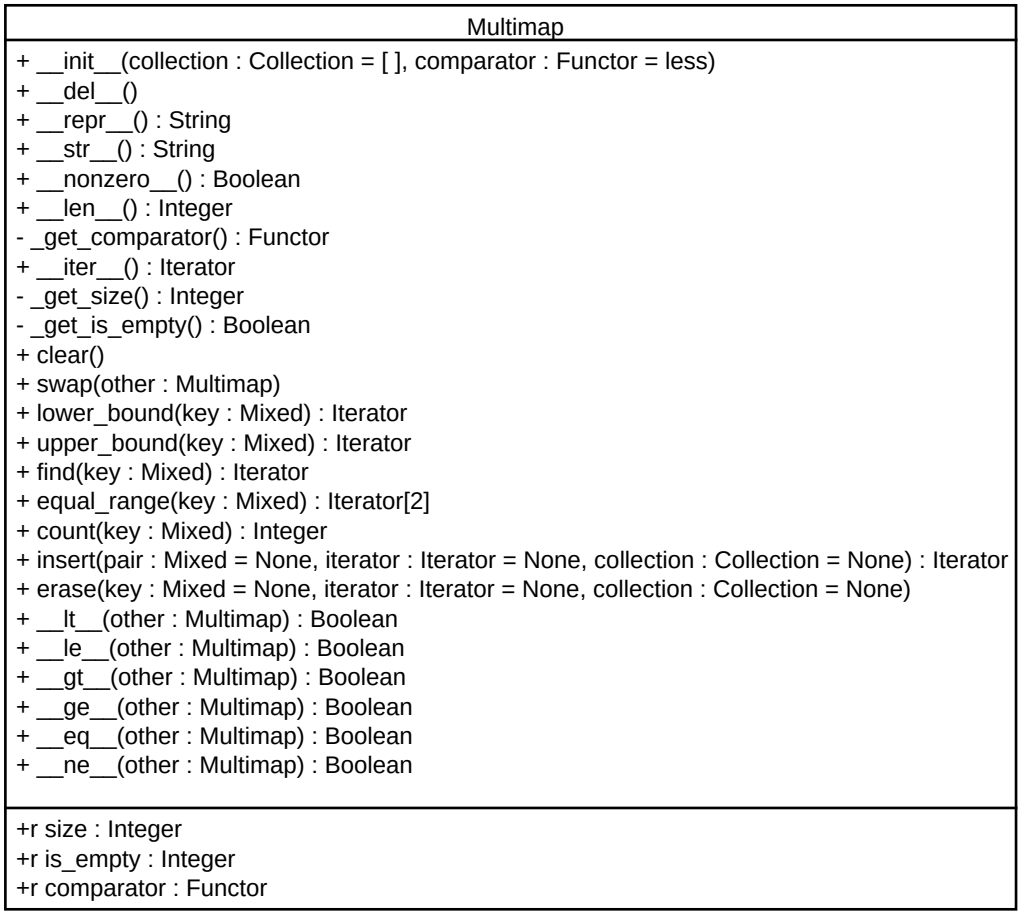
I en komplet implementering af et multikort bør det ikke være muligt at ændre i nøglen i et par, men at ændre i elementet i et par er muligt, når det først er lagt i kort. Dette ville kunne ændre parrenes orden i kortet.

Figur 3.19 (s. 50) viser `UML`-diagrammet for en klasse af multikortkonceptet og figur 3.18 (s. 49) de før omtalte egenskaber for konceptet.

Konceptet kan implementeres som en adapter af en skipliste eller et binært søgetræ. Det kunne menes, at konceptet kunne implementeres som en adapter til multisætkonceptet, men her er det vigtigt at huske, at en korrekt multisæt implementering ikke bør lade det være muligt at rette i dets elementer, og da vil det ikke være muligt at rette i et par, fordi parret er et element i et sæt, og dermed forsvinder den før omtalte mulighed for at kunne rette i elementdelen af et par.



Figur 3.18: Multikortkonceptets egenskaber. Sammenligningsfunktionen (n) tager nøgledelen i to par (P_1 og P_2) og sammenligner dem, hvis det første pars nøgle er før det andet i ordenen returneres sandt. Egenskaber, der returnerer en iterator, er vist med parret denne, iterator peger på.



Figur 3.19: UML-diagrammet for multikortkonceptet.

3.14 Prioritetskø

Prioritetskøkonceptet (*priority queue*) er fra C++0x specifikationen [2] af det samme koncept. I C++0x er en Prioritetskø en adapter af en vektor eller deque og kan ikke itereres, men i CPH collections er prioritetskøen en selvstændig beholder og kan itereres. Metoderne i dette koncept er beskrevet i bilag B.10 (s. 143).

Prioritetskøkonceptet er en udvidelse af beholderkonceptet og har følgende egenskaber udover dem i beholderkonceptet:

Sammenligning Det er muligt at få sammenligningsfunktionen, men denne kan kun sættes ved oprettelsen.

Udtræk Det er muligt at få udtræksfunktionen, men denne kan kun sættes ved oprettelsen.

Top Det er muligt at få eller fjerne det øverste element.

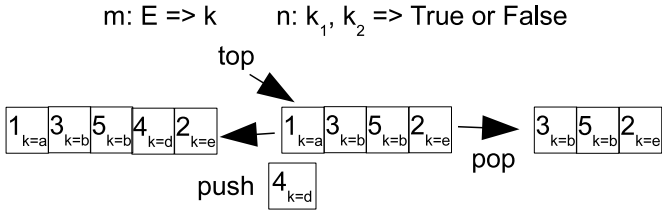
Indsætning Det er muligt at indsætte et element, prioritetskøen finder selv det korrekte sted i køen, der opretholder dens orden.

Ordenen i konceptet er, at gives sammenligningsfunktionen udtrækket fra to på hinanden følgende elementer så returneres sandt.

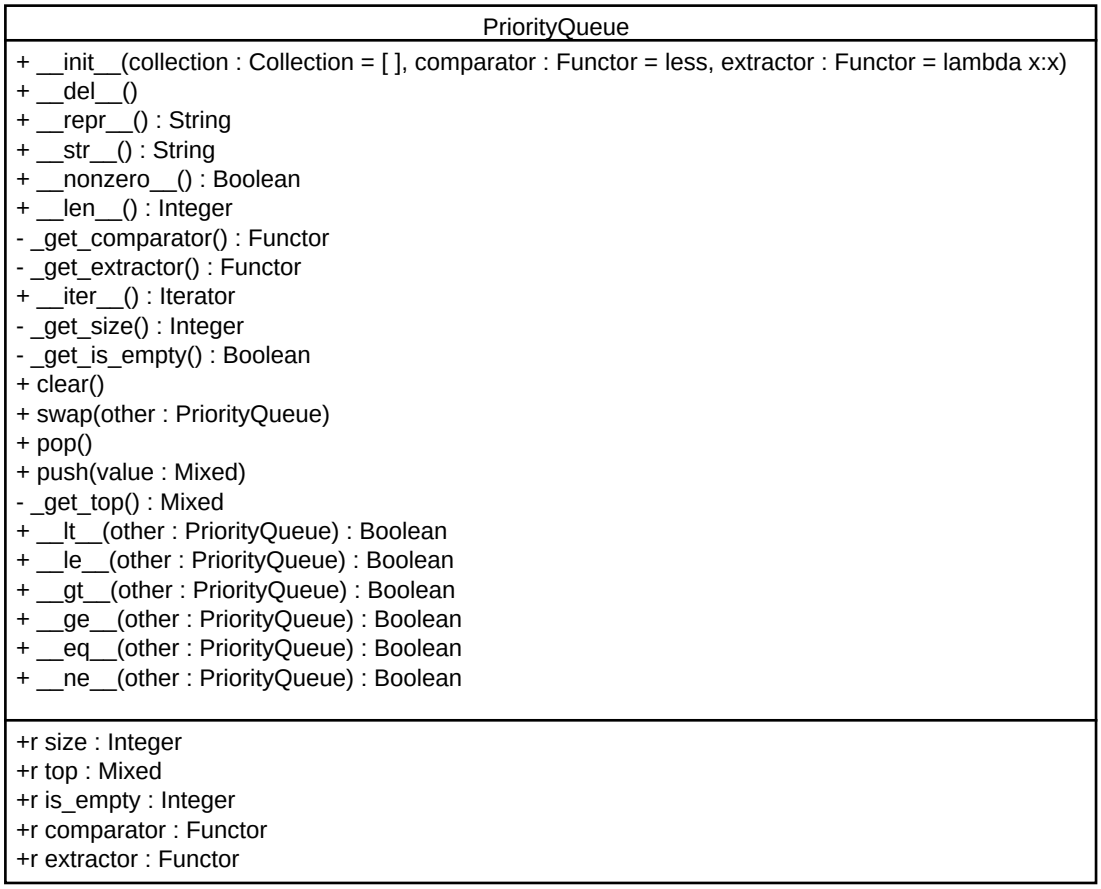
I en komplet implementering af en prioritetskø bør det ikke være muligt at ændre i et element, sammenligningsfunktionen eller udtræksfunktionen. Dette ville kunne ændre ordenen i prioritetskøen.

Figur 3.21 (s. 52) viser UML-diagrammet for en klasse af prioritetskøkonceptet og figur 3.20 de før omtalte egenskaber for konceptet.

Konceptet kan implementeres som en adapter af en skipliste eller et binært søgetræ.



Figur 3.20: Prioritetskøkonceptets egenskaber. Udtræksfunktionen (m) tager et element (E) og returnerer dets nøgle (k) og sammenligningsfunktionen (n) tager to nøgler (k_1 og k_2) og returnerer sandt, hvis den første nøgle er før den anden i ordenen. Egenskaber, der returnerer en iterator, er vist med elementet, denne iterator peger på.



Figur 3.21: UML-diagrammet for prioritetskøkonceptet.

Kapitel 4

Basisimplementeringer

Dette kapitel beskriver basisimplementeringer af strukturer og koncepter. En basisimplementering er en impletering, der ikke findes designmønstre der beskriver den eller en implementering af en specifik struktur. Den første del af kapitlet vil beskrive interne strukturer og den anden del vil beskrive implementeringer af koncepter, der ikke kunne beskrives som brugen af et kendt designmønster.

4.1 Interne strukturer

4.1.1 Nodetyper

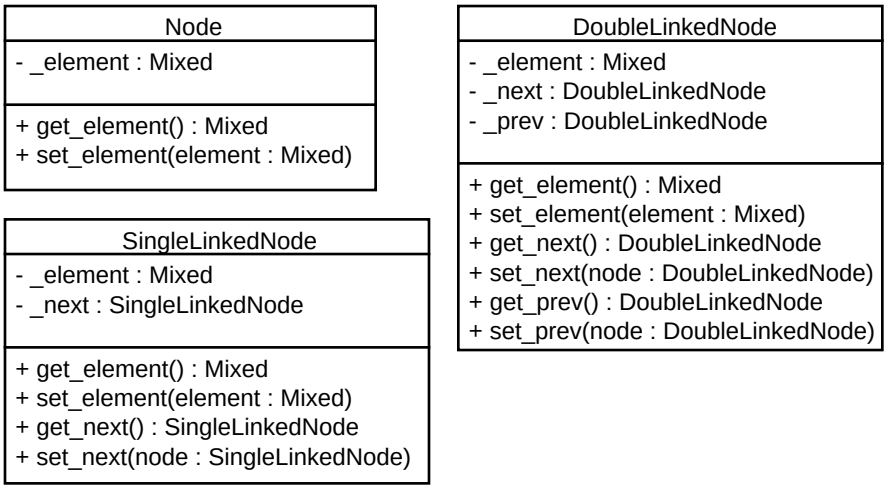
Basis nodetyper

Denne sektion beskriver nodestrukturene, der findes i den nuværende implementering af CPH collections. Noder er interne strukturer, der bruges i realisationer af visse af beholderkoncepterne. En node er en datatype, som holder på ét element og eventuelt peger på én eller flere andre noder. En enkeltlinket node kan pege på én anden node, og en dobbeltlinket node kan pege på to andre noder. Figur 4.1 (s. 54) viser klasserne for basis noden og dobbelt- og enkeltlinket noderne.

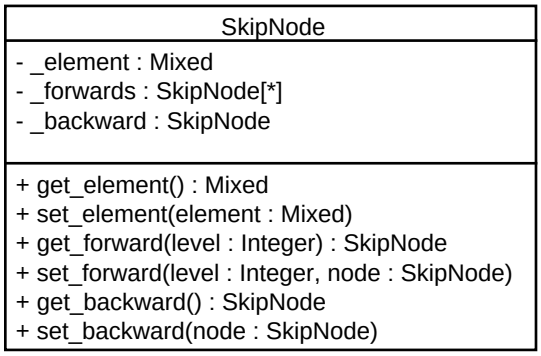
Det er bevidst, at noderne ikke bruger Python-egenskaber til at indkapsle hent og sæt metoderne. Grunden til dette er, at eftersom noderne er interne, er de ydelse optimeret. (Som sektion 2.2.1 (s. 19) viser, er det tidsbesparende at undgå Python-egenskaber.)

Skipnode

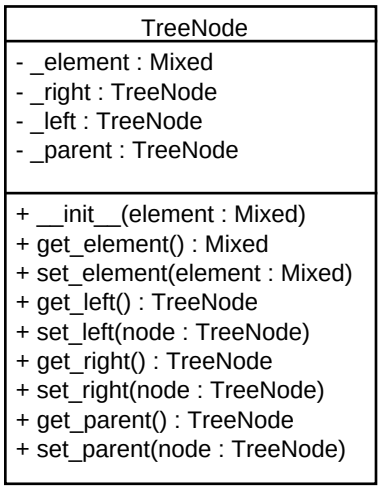
En skipnode er en nodetype, der peger på flere noder, som mulige næste noder og på en enkelt node som den foregående. Antallet af noder, som skipnoden peger på som de næste, er dens niveau (*level*). Figur 4.2 (s. 54) viser klassen for denne node.



Figur 4.1: Klassediagram af nodetyper.



Figur 4.2: Klassediagram for Skipnode.



Figur 4.3: Klassediagram for TreeNode.

Trænode

En trænode er en nodetype med tre pegere. To af pegerne peger på henholdvis det højre og venstre barn. Den sidste peger på noden, der er forældre til den pågældende node. Figur 4.3 (s. 55) viser klassen for denne node.

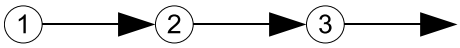
4.1.2 Nodestrukturer

Denne sektion beskriver nodestrukturer. Dette er ikke beskrivelser af klasser, men en beskrivelse af, hvordan de før omtalte noder kan bruges i opbygningen af sammenlinket strukturer.

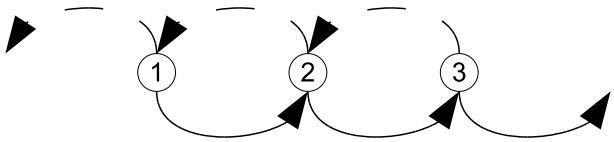
Linket lister

Figur 4.4 (s. 56) viser en enkeltlinket liste, der gør brug af de før nævnte enkeltlink noder. Hver node i listen peger på den efterfølgende node undtagen den sidste node, der ikke peger på en efterfølgende node. En enkeltlinket liste kan itereres fra den første node til den sidste node, men ikke fra den sidste node til den første node. Noder kan indsættes efter en tilfældig node i listen i konstant tid men ikke før. (Forudsat den tilfældige node kendes.) Det er ikke muligt at fjerne en tilfældig node i listen i konstant tid. (For at fjerne en node skal den foregående node sættes til at pege på den efterfølgende, og for at finde den foregående skal der itereres til denne.) Hvis den første node i listen kendes, er det muligt at indsætte en node foran denne i konstant tid og at fjerne den første node i listen i konstant tid.

Figur 4.5 (s. 56) viser en dobbeltlinket liste. Denne liste gør brug af de før nævnte dobbeltlink noder. Hver node i listen peger på den foregående og



Figur 4.4: Enkeltlinket liste af noder.



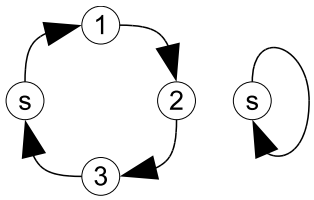
Figur 4.5: Dobbeltlinkednode liste af noder.

efterfølgende node undtagen den første og sidste node, der henholdsvis ikke peger på en foregående og en efterfølgende node. En dobbeltlinket liste kan itereres fra den første til den sidste node og fra den sidste til den første node. Det er muligt at indsætte en node før og efter en tilfældig node i listen i konstant tid og det er muligt at fjerne en tilfældig node fra listen i konstant tid.

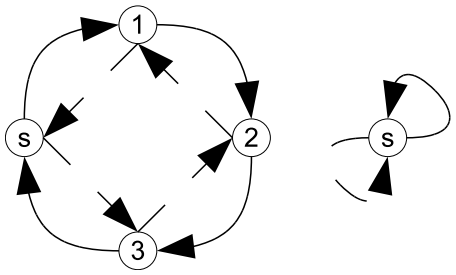
For både den enkelt- og dobbeltlinket liste gælder det, at der skal tages specielle hensyn når, der ikke er nogen noder i listen. Dette kan undgås, hvis der bruges en speciel node (*sentinel*).

For en enkeltlinket liste peger den specielle node på den første rigtige node i listen som dens efterfølgende, og den sidste node i listen peger på den specielle node som dens efterfølgende. Hvis listen er tom, peger den specielle node på sig selv. For en dobbeltlinket liste gælder de samme regler som ved den enkeltlinket liste, og derudover peger den specielle node på den sidste node i listen som dens foregående node, og den første node i listen peger på den specielle node som den foregående. Hvis listen er tom, peger den specielle node på sig selv som den foregående node.

Brugen af den specielle node danner enkelt- og dobbeltlinket ringe, som ses i figurene 4.6 og 4.7 (s. 57). Fordelen ved ringene fremfor listerne er først og fremmest at de altid har en node (den specielle node) og i Python's iteratorkoncept kan den specielle node bruges som start og ende node. (Python iteratorer starter før det første element.)



Figur 4.6: Enkeltlinket ring med og uden noder; den speciele node er s.



Figur 4.7: Dobbeltlinket ring med og uden noder; den speciele node er s.

Figur 4.8 (s. 58) viser en skipliste [11]. En skipliste er en sorteret liste, hvor hver node i listen er en skipnode. I hvert niveau i en node peges på den næste node i listen, der mindst har det samme niveau. En nodes niveau bestemmes ved indsættelsen af et element, hvordan dette niveau bestemmes er afhængig af implementeringen. I CPH collections implementering tildeles niveaut med en tilfældighedsfunktion.

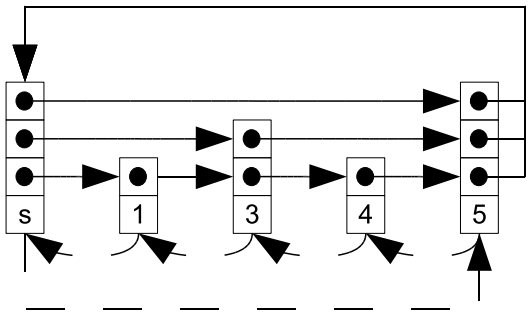
Tilfældigheden bruges til at mindske sandsynligheden for, at en node får et højere niveau. Tilfældigheden brugt i CPH collections er 50%, hvilket vil sige, at 50% af noderne med mindst niveau n også er en node med niveau $n+1$. (Alle noder har mindst niveau èt.) Denne opbygning medfører, at et skridt taget i et niveau mellem to noder springer noderne over, der har et lavere niveau. Effekten af opbygninger er, at søgninger i listen kan gøres hurtigere i forhold til en linket liste.

For at gennemløbe alle noderne i listen besøges noderne gennem deres niveau 1 link. Listen kan også itereres baglæns eftersom, at hver node peger på den foregående node. Listen starter med en speciel node, der peger frem ad til de næste noder og baglæns til den foregående node. Disse to ting (den specielle node og baglænspegerne) er tilføjelser i forhold til skiplisten i [11].

Det gennemsnitlige antal af pegere en node i listen har er 3; En node har 1 baglæns peger og $1/(1-p)$ [11] (tabel 1) forlæns pegere, hvor p er sandsynligheden brugt i genereringen af nodeniveaur. I CPH collections er $p = \frac{1}{2}$ og max nivauet en node kan have er sat til 11. Dette betyder at det forventede gennemsnit af pegere kun er sandt, så længe antallet af elementer er under 2^{11}. Stiger antallet over dette falder gennemsnittet af pegere, men det samme gør effektiviteten af operationer også.

Eftersom at skipnoden i CPH collections holder forlænspegerne i Python lister, er den reelle pladsforbrug brugt til forlænspegere altid større end det forventede gennemsnit, da Python lister har plads til flere elementer end de indeholder.

Figur 4.9 (s. 59) viser skipliste klassen, der indkapsler noderne i skiplisten. I forhold til listen beskrevet i [11] tager indsæt og fjern metoderne en iterator og klassen har derudover metoderne *lower_ bound* og *upper_ bound*. Disse tilføjelser er grundet implementeringens brug i for eksempel sæt implementeringer. Itera-



Figur 4.8: Skipliste; noden med element s er den specielle node.

toren sendt til listen i både indsættelse og fjern metoderne har ingen effekt, men er der for at kunne virke med for eksempel en sæt implementering.[1] Med metoderne *lower_bound* og *upper_bound* kan en hurtigere søgning efter en nøgle foretages ved brug af skiplistens opbygning end ved et gennemløb af alle noderne efter nøglen.

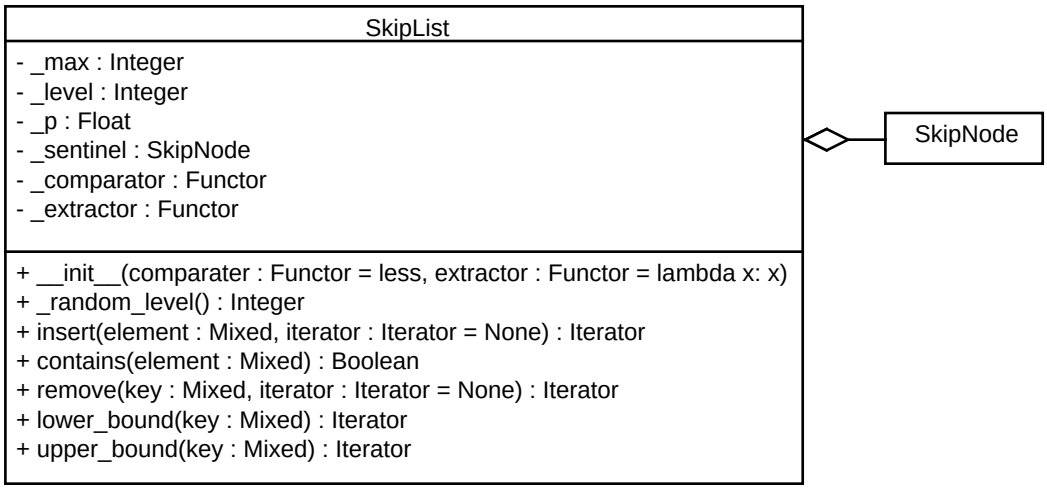
4.1.3 Binært søgetræ

Figur 4.10 (s. 59) viser et binært søgetræ (*binary search tree*). Et binært søgetræ er et træ bestående af trænoder, hvor funktionen f bestemmer træets struktur. Funktionen f givet to elementer a og b returnerer sandt eller falsk. Følgende regler gælder for en node i træet med element a:

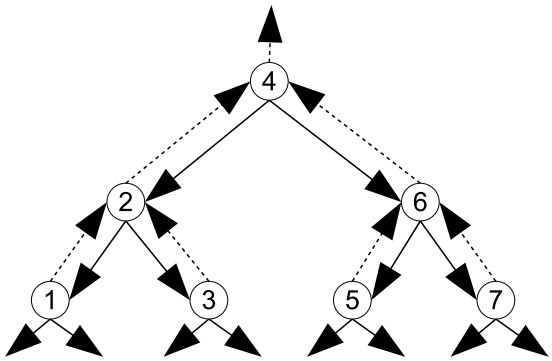
- Element b i det venstre barn resulterer i $f(b, a) => sandt$. (Hvis der peges på et venstre barn.)
- Element b i det højre bar resulterer i $f(b, a) => falsk$. (Hvis der peges på et højre barn.)
- Hvis noden ikke peger på nogen forældrenode er denne node roden.

Denne opbygning betyder, at de fleste operationer, som for eksempel søgning, kan udføres med en kompleksitet baseret på træets højde i stedet for antallet af elementer i træet. Et gennemløb af elementerne i træet fra det første til det sidste vil være linær i antallet af elementer. (Hver gren i træet bliver besøgt to gange, én gang for at komme fra forældren til barnet og én for at komme fra barnet til forældren). Dog vil hver iteration ikke kræve det samme antal

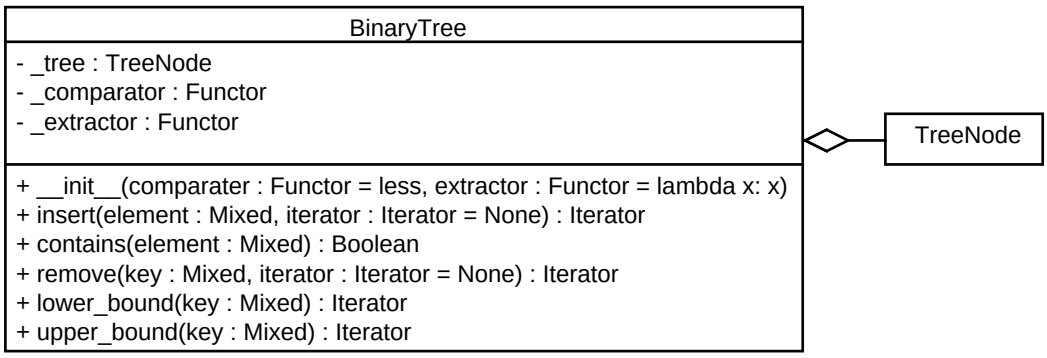
[1] Ideen med at give en iterator er, fra sættes side, at implementeringer kan bruge iteratoren til mere effektivt at finde stedet, hvor der skal indsættes eller fjernes et element. Elementet, der skal indsættes eller fjernes, antages at være et sted efter elementet iteratoren er ved. Dette ville være muligt i skipliste, hvor iteratoren peger på en node i listen, men, hvis noden for det nye element får et højere niveau end noden iteratoren er ved og der ikke findes en node med det samme eller højere niveau end den nye node fra den givet node til stedet den nye skal indsættes. Skal en node før den givet node findes.



Figur 4.9: Klassediagrammet for en skipliste.



Figur 4.10: Binært søgetræ; de stiplede pile er pegere fra barn til forældre og de andre fra forældre til barn.



Figur 4.11: Klassediagrammet for et binært søgetræ.

operationer. I figur 4.10 (s. 59) kan dette ses ved, at der kræves et skridt for at komme fra element 2 til element 3, men to skridt for at komme fra element 3 til element 4. I et velbalanceret træ, som det i figur 4.10 (s. 59) er højden af træet $\lg n$, men i værste tilfælde kan træet blive til en linket liste af noder og så er dets højde n. Dette kan dog modvirkes med forskellige teknikker, som for eksempel rød-sort træ (*red black tree*), der opretholder træets balance.

Figur 4.11 viser `BinaryTree` klassen, der indkapsler noderne i træet. Iteratoren sendt til træet i både indsættelse og fjern metoderne har ingen effekt, men er der for at kunne virke med for eksempel en sæt implementering. Metoderne *lower_bound* og *upper_bound* er der for at bruge træets struktur til en hurtigere søgning efter en nøgle i stedet et gennemløb af alle elementerne, indtil nøglen findes.

4.2 Implementeringer af koncepter

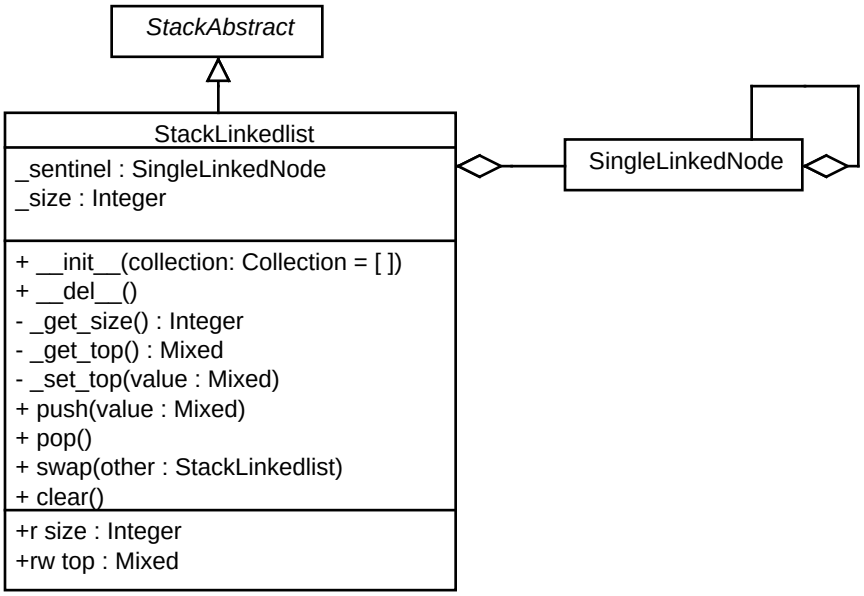
4.2.1 Stak implementeret med enkeltlinket ring

En mulig implementering af en stak er at bruge en enkeltlinket ring af noder.

Metoden `push` opretter en ny node med det givet element; den nye node sættes til at pege på noden som den specielle node peger på, og den specielle node sættes til at pege på den nye node. Metoden `pop` sletter elementet i den første node og sætter den specielle node til at pege på noden, den første node peger på.

Eftersom at en enkeltlinket ring ikke har nogen tæller for antallet af elementer, så tæller stak implementeringen dette.

Eftersom at det øverste element i stakken er elementet i den første node, så vil en forlæns iteration starte i denne node og for hver iteration gå til den næste node i ringen. Figur 4.12 (s. 61) viser klassediagrammet for denne implementering.



Figur 4.12: Klassediagrammet for stak implementeret med enkeltlinket ring.

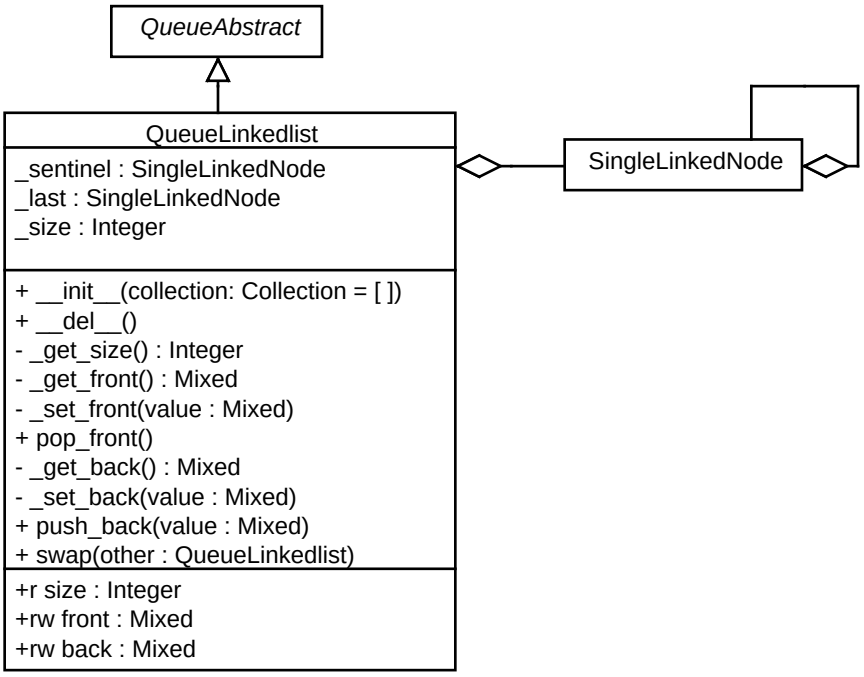
4.2.2 Kø implementeret med enkeltlinket ring

En mulig implementering af en kø er at bruge en enkeltlinket ring af noder.

Metoden `push_back` opretter en ny node med det givet element; den nye node sættes til at pege den specielle node i ringen og en peger til den nye node huskes. Metoden `pop_front` sætter den specielle node til at pege på den node som noden den specielle node peger på. Dette fjerner referencen til noden den specielle node pegede på medmindre køen var tom. Python burde selv slette noder, der ikke refereres til.

Eftersom at den enkeltlinket ring ikke har nogen tæller for antallet af elementer, så tæller kø implementeringen dette.

Eftersom at det første element i køen er elementet i noden den specielle node peger på, så vil en forlæns iteration starte i denne node og for hver iteration gå til den næste node i ringen. Det er nødvendigt at have en peger til den sidste node for at kunne indsætte en node efter denne og kunne få elementet i noden. Figur 4.13 (s. 62) viser klassediagrammet for denne implementering.



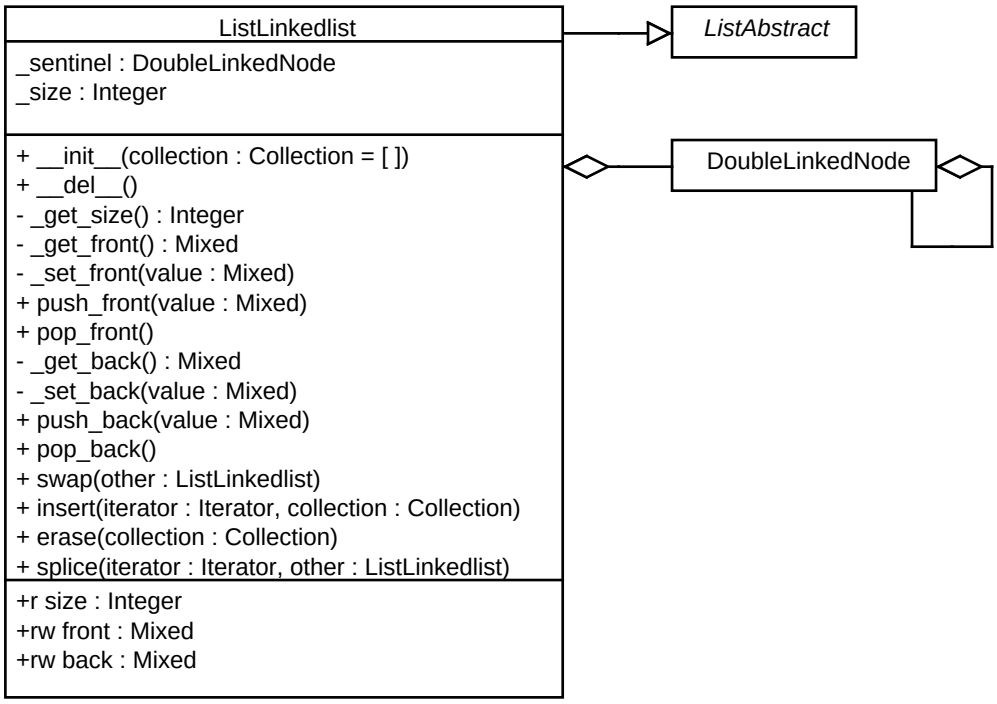
Figur 4.13: Klassediagrammet for kø implementeret med enkeltlinket ring.

4.2.3 Liste implementeret med dobbeltlinket ring

En mulig implementering af en liste er at bruge en dobbeltlinket ring af noder. Den specielle node peger på den første og sidste node i listen. Metoderne der fjerner er element fra en af enderne af listen implementeres ved, at den specielle node sættes til at pege på noden efter eller før noden der fjernes og den nye første eller sidste node sættes til at pege på den specielle node. Ligeledes kan noder indsættes i hver enden af listen ved at rette på linkene mellem den specielle node, den nye node og den gamle node, som den specielle node pegede på.

Da en dobbeltlinket ring ikke tæller antallet af elementer i ringen tælles dette i implementeringen af listen. Figur 4.14 (s. 63) viser klassediagrammet for denne implementering.

Eftersom at toendetkøkonceptet i CPH collections i princippet er listekonceptet, hvor fjernelse og indsættelse af elementer kun er tilladt i enderne, så kan en toendetkø også implementeres med en dobbeltlinket ring.



Figur 4.14: Klassediagrammet for liste implementeringen med dobbeltlinket ring.

Kapitel 5

Anvendelse af designmønstre

Dette kapitel gennemgår de designmønstre (*design patterns*), der bruges i CPH collections. Disse designmønstrene er hovedsagligt beskrevet i [6]. Designmønstre bruges til at vise, hvordan én funktionalitet kan implementeres i et program. Disse mønstre bliver fundet via erfaring; med dette menes, at hvis en funktionalitet bliver implementeret i flere programmer så vil, der før eller senere, opstå en viden om den bedste teknik (*best practise*) at implementere den givne funktionalitet på. Det er denne viden som designmønstre bruges til at beskrive. Den efterfølgende liste beskriver visse af de elementer, som hører med til beskrivelsen af designmønstre ifølge [6]:

Navn: Et unikt og sigende navn foranlediger, at navnet bliver informationsbærende for mønstret og kan derfor bruges som reference til mønstret.

Formål: Beskriver funktionaliteten designmønstret dækker over.

Aliaser: Hvis mønstret er kendt under andre navne, så skrives disse i denne del af beskrivelsen.

Motivation: En forklaring af det problem, som mønstret løser.

Struktur: Grafiske illustrationer af mønstret; i tilfælde af objektorienterede designmønstre, bør dette som minimum inkludere et klassediagram.

Konsekvenser: Hvad opnås der ved brugen af mønstret både de positive effekter, men også de negative.

Implementeringer: Eksempler på, hvordan det givne mønster kan implementeres; ideelt set vist i flere programmeringssprog.

Kendte brugstilfælde: Eksempler på, hvor det givne mønster er blevet brugt. Eftersom et designmønster er baseret på bedste teknik efter erfaring må det have eksempler på reel brug.

Relaterede mønstre: Designmønstre, der opnår stort set samme resultat, eller som bliver brugt sammen med det givne mønster. I det første tilfælde beskrives, hvornår det andet mønster skal bruges, og i det andet tilfælde skal der beskrives, hvornår de to mønstre bruges i et samarbejde.

I tabel 5.1 beskrives formålet med de designmønstre, der er brugt i CPH collections. Der findes ikke beskrivelser af designmønstrer, som svare til den ovenstående liste, i speciealet.

Navn	Formål
Iterator [6]	Implementerer en ensartet grænseflade for gennemløb af elementerne i forskellige datastrukturer. Mønstret lader det dermed være muligt at abstrahere fra, hvordan et sådan gennemløb reelt foretages. Når grænsefladen for gennemløb af elementer er ens, kan algoritmer implementeres, så de bruger den generiske iteratorgrænseflade og dermed selv bliver generiske.
Skabelon-metode [6]	At minimere den del af en metode, der skal implementeres, ved implementering af metodens konstante dele som en metode i en superklasse og refaktorere de variable dele af metoden ud som kald til andre metoder, der er abstrakte i superklassen, men konkrete i underklasser.
Abstrakt superklasse-grænseflade	At definere grænsefladen som klasser, der arver fra den abstrakte superklasse skal implementere. Dette gør, at algoritmer kan implementeres så de gør brug af grænsefladen defineret i den abstrakte klasse og kun skal verificere, at et objekt er en instans af en klasse, der arver fra denne klasse.
Adapter [6]	Hvis der findes en eksisterende datastruktur, der kan bruges til at implementere en datatype, men denne datastruktur ikke har typens grænseflade, så kan adaptermønstret bruges. Mønstret bruges ved, at der designes en klasse (adapteren), der overholder grænsefladen for den ønskede datatype, men som rediregerer kald til en instans af den eksisterende datastruktur (den adapterede).

Bro [6]	Dette mønster lader det være muligt at abstraktion og implementering kan variere uafhængig af hinanden. Abstraktionen er den grænseflade som brugeren har og denne kan varieres via nedarvning fra abstraktionen. Implementeringen er, hvordan en løsning reelt er implementeret, En abstraktion har en aggregering til en implementering. Ved udskiftning af af denne instans varieres implementeringen. Hovedforskellen mellem bro- og adaptermønstret er, at adaptermønstret bruges mod eksisterende implementeringer og bromønstret bruges, når det ønskes at kunne variere abstraktion og implementering. Hvordan abstraktionen variere implementeringen afhænger af implementeringen af abstraktionen, der implementeres til enten selv styre variering eller lader det være op til brugeren.
Klassefabrik	Dette mønster lader brugeren abstrahere væk fra de individuelle realisationer af en datatype. Dette opnås ved implementering af en klassefabrik, der når en instans af den bliver efterspurgt, i stedet returnerer en instans af en af realisationerne. Hvilken realisation, der returneres, kan brugeren vælge via argumenterne, der gives til klassefabrikken[1].
Dekorator [6]	At udvide en implementering (den dekorerede) med en funktionalitet, der ikke er i den, ved omkapsling af en anden implementering (dekoratoren). Dekoratoren har samme grænseflade som den dekorerede, men dens metoder opfylder den nye funktionalitet og kalder den dekorerede des tilsvarede metoder. (Medmindre den nye funktionalitet betyder, at de ikke skal kaldes.)

Tabel 5.1: De anvendte designmønstre og deres formål.

Designmøstre vil i de efterfølgende sektioner blive beskrevet delvist med `UML`-diagrammer. Her er det vigtigt at huske, at i Python erklæres typerne på argumenter eller varabler i objekter ikke. Dermed vil variabler og argumenter kun skulle forventes at overholde en grænseflade og ikke et arvehieraki.

5.1 Iteratormønstret

Denne sektion beskriver de konkrete anvendelser af iteratormønstret i CPH collections. Men først vil der blive beskrevet, hvordan iteratormønstret bliver implementeret i Python.

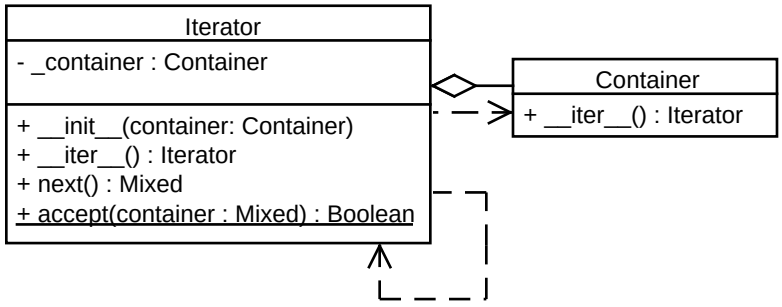
I Python implementeres iteratorer ved, at en klasse implementerer metoderne `next` og `__iter__` [8]; instanser af en sådan klasse bruges som iteratorer.

[1] Eftersom, at det ikke er muligt at oprette en instans af en klassefabrik, så er dette ikke et eksempel på brug af én instansmønstret (*singleton pattern*).

next: Rykker iteratoren til det næste element i en beholder og returnerer dette, men er iteratoren ved det sidste element i iterering rejses undtagelsen `StopIteration`. Dermed starter en Python iterator på ingenting og slutter på det sidste element i itereringen. Dette forskelligt fra iteratorer i C++ der starter på det første element og slutter efter det sidste element.

`__iter__`: Returnerer iteratorinstansen selv[2].

En beholderklasse implementerer metoden `__iter__` til at returnere en instans af en iteratorklasse oprettet med beholderen selv som argument. Klassediagrammet for dette ses i figur 5.1.



Figur 5.1: Iteratormønstrets klasseediagram.

Hvilke værdier en iteratorenimplementering gemmer i variabler er afhængig af iteratoren. Denne opbygning gør, at algoritmer, der foretager gennemløb af elementerne i en beholder og bruger værdien af disse elementer, implementerer for-løkken i listing 5.1.

Listing 5.1: Iteration af en beholder.

```
1 container = Container()
2 for e in container:
3     print e
```

Hvis variablen *container* i eksemplet er en beholder med n elementer, så vil for-løkken udskrive de n elementer. Forudsat at klassen, som *container* er en instans af, implementerer metoden `__iter__` til at returnere en instans af en iteratorklasse, der ved en fuld iterering returnerer alle n elementer i beholderen før undtagelsen `StopIteration` rejses. `For`-løkken fanger selv denne undtagelse og ved denne afbrydes iterationen, men undtagelsen bliver stoppet fra at rejses videre. (Dette er grunden til undtagelsen ikke har `Error` i slutningen af sit navn, da det er en forløbssundtagelse (*control flow exception*) og ikke en fejlundtagelse.) Som det ses af ovenstående kodeeksempel abstraherer Python ikke

[2] At iteratorer implementerer en metode, der retunerer dem selv, kan virke meningsløs, men antagligt skyldes det implementeringen af for-løkker i Python.

alene væk fra, hvordan et gennemløb af en beholder foretages, men også væk fra iteratorgrænsefladen.

Klassemetoden `accept` findes på alle iteratorer i CPH collections, grunden til dette er iteratorfabrikken, se sektion 5.6.2 (s. 93).

I Python er det muligt at implementere funktioner, så de kan bruges som iteratorer. Disse funktioner kaldes for generatorer (*generators*). En generator [10] er en funktion, der når den bliver kaldt, returnerer et generatorobjekt. Generatorobjektet implementerer iteratorgrænsefladen. Når metoden `next` kaldes, eksekveres koden fra det nuværende sted i funktionskoden til det næste sted i koden, hvor nøgleordet `yield` står. Værdien efter nøgleordet returneres til kalderen af metoden `next` og funktionens tilstand huskes af generatorobjektet. Hvis funktionen når til en `return`-erklæring, rejses undtagelsen `StopIteration` af metoden `next`. En generatorfunktion kan dermed bruges som en iterator som vist i listing 5.2. Eksemplet viser en generatorfunktion der, når den gives en beholder med metoderne `__len__` og `__getitem__`, starter en iterering over beholderens elementer og for hver iteration returneres et element.

Listing 5.2: Generator for direkte indekserbar beholder.

```
1 def iterator(container):
2     for i in xrange(len(container)):
3         yield container[i]
```

Generatorfunktionen kan da bruges i gennemløbet af beholderen som vist i listing 5.3:

Listing 5.3: Brug af generator.

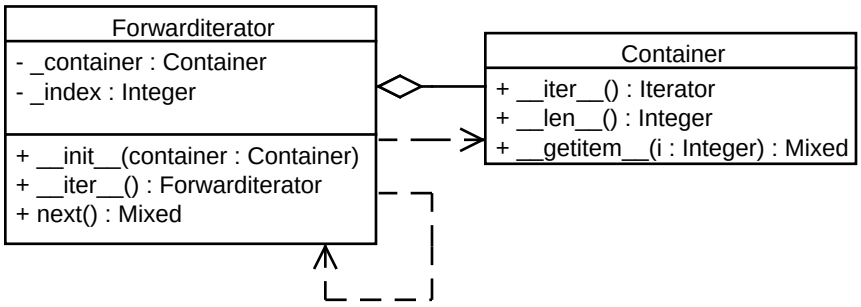
```
1 for e in iterator(container):
2     print e
```

Der findes to gængse former for iterator koncepter. (forlænsiterator og baglænsiterator.)

5.1.1 Forlænsiterator

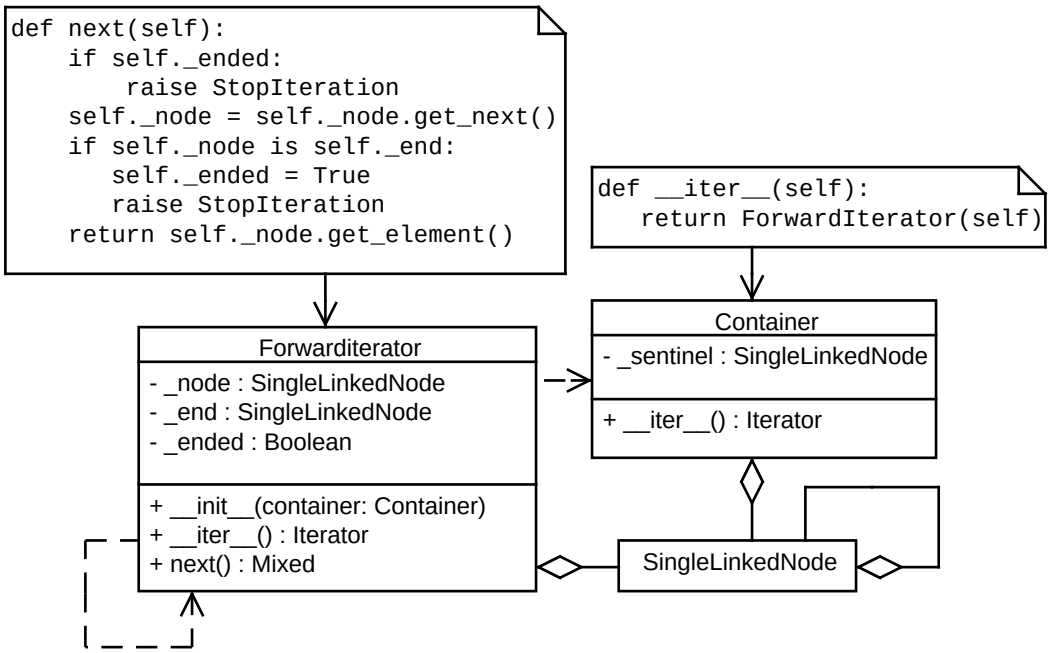
En forlænsiterator (*forward iterator*) implementerer en iterering over elementerne i en beholder, som menes at være fra det første element i beholderen til og med det sidste element i beholderen.

For en beholder, der implementerer direkte indeksering (Python-list, vektor), og, som har en metode, der returnerer antallet af elementer i beholderen; kan en forlænsiterator implementeres, ved at den starter ved indeks -1 og for hver iteration lægges én til indekset og elementet i det nye indeks i beholderen returneres, men hvis indeks er blevet lig med antallet af elementer i beholderen, rejses undtagelsen *StopIteration*. Figur 5.2 (s. 69) viser klassediagrammet for, hvordan iteratorklassen og beholderklassen ser ud.



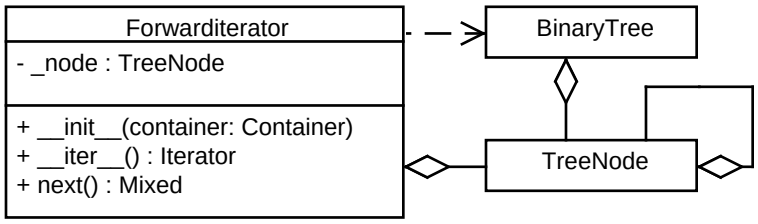
Figur 5.2: Forlænsindeksiterator- og beholderklassen.

En implementering af en forlænsiterator for en enkeltlinket eller dobbeltlinket ring implementeres ved, at den oprettes med den specielle node, og for hver iteration rykker iteratoren til den næste node i ringen; tester om denne node er den specielle, hvis den er, rejses `StopIteration` undtagelsen ellers returneres elementet gemt i noden. Hvis `StopIteration` er blevet rejst skal alle efterfølgende forsøg på iterering også rejse undtagelsen, og det er derfor nøvendigt for iteratoren at huske, om dette er gjort. Klassediagrammet for den iterator kan ses i figur 5.3 (s. 70), og i noterne kan ses, hvordan iteratorklassen og beholderklassen for dette ser ud. En skipliste kan principielt itereres på samme måde, men i stedet for at iteratoren kalder `get_next` kaldes `get_forward` med 0 som argument.

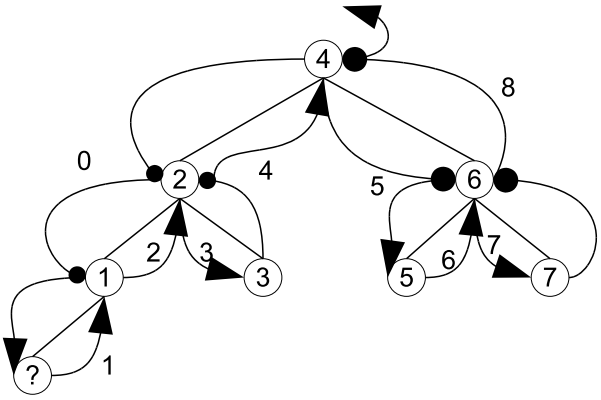


Figur 5.3: Klassediagram for forlænsnodeiterator og beholder; noterne viser hvordan iteratoren oprettes fra beholderen og itererer over beholderens noder.

En implementering af en forlænsiterator til et binært søgetræ, kan implementeres ved at oprette en falsk trænode, hvis forældre er noden mest til venstre i træet. Ved iterering tester iteratoren om der er et højre barn, hvis der er itereres til dette og så langt som muligt ned i barnets undertræ kun ved at tage venstre børn, ellers, hvis der ikke er noget højre barn, rykkes til nodens forælder indtil den node, iteratoren kommer fra er, forælderens venstre barn. Elementet i noden, der stoppes ved, returneres, hvis iteratoren ikke er ved en node rejses `StopIteration`. (Den har taget forælderen til toppen af træet.) Figur 5.4 (s. 71) viser klassediagrammet for dette, og figur 5.5 (s. 71) viser stien for en sådan iterator.



Figur 5.4: Forlænstrænodeiterator-, beholderklassen.



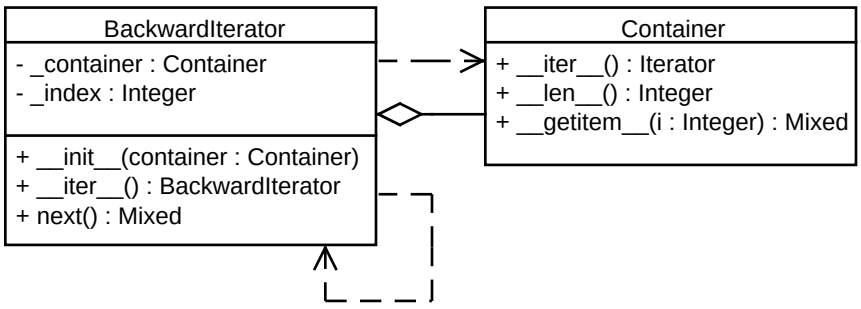
Figur 5.5: Forlænsiterering af et binært søgetræ. Tallene på pilene indikerer rækkefølgen af iterationerne, en iteration kan besøge en node på vejen til en anden, dette er indikeret med en sort cirkel. Noden med ?-tegnet er den falske trænode.

5.1.2 Baglænsiterator

En baglænsiterator (*backward iterator*), implementerer en iterering over elementer i en beholder fra det sidste element i beholderen til det første element i beholderen.

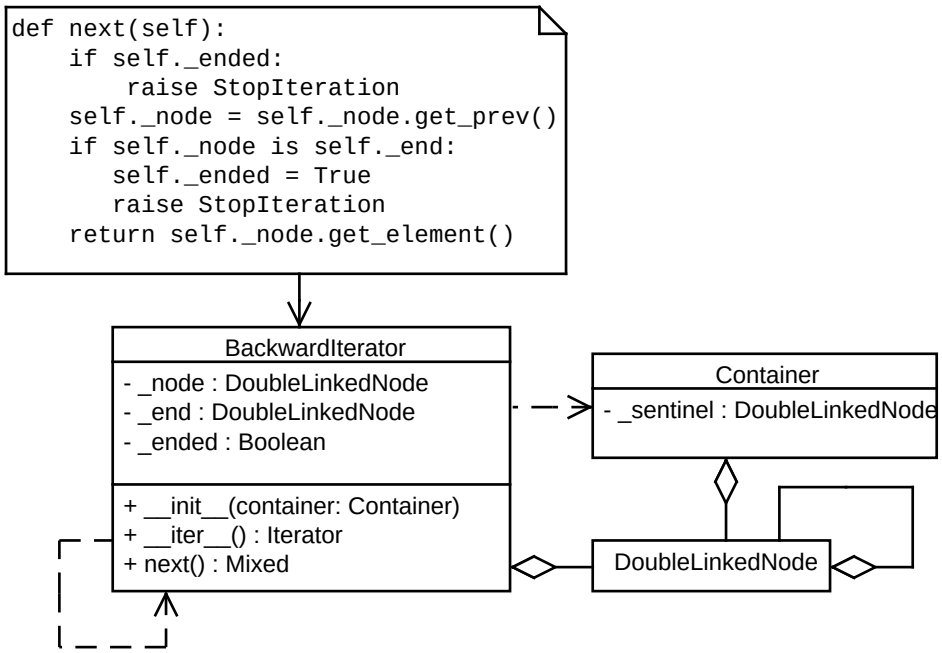
For en beholder, der implementerer direkte indeksering (Python-list, vektor) og har en metode, der returnerer antallet af elementer i beholderen; kan en baglænsiterator for beholderen implementeres ved at starte ved indekset efter det sidste element og for hver iteration trække én fra dette indeks. Elementet i det nye indeks returneres fra beholderen medmindre indekset er -1, hvilket betyder iteratoren er nået enden og derfor rejses undtagelsen `StopIteration`. (Når `StopIteration` rejses ved indeks -1 antages det, at beholderens indeks starter ved 0.) Figur 5.6 (s. 72) viser klassediagrammet for iterator- og beholder-

klassen.



Figur 5.6: Baglænsindeksiterator- og beholderklassen.

En implementering af en baglænsiterator for en dobbeltlinket ring implementeres ved, at den oprettes med den specielle node, og for hver iteration rykker iteratoren til den foregående node i ringen; tester om denne node er den specielle, hvis den er, rejses `StopIteration` undtagelsen ellers returneres elementet gemt i noden. Hvis `StopIteration` er blevet rejst skal alle efterfølgende forsøg på iterering også rejse undtagelsen, og det er derfor nøvendigt for iteratoren at huske, om dette er gjort. Klassediagrammet for iteratoren kan ses i figur 5.7 (s. 73), og i noterne kan ses, hvordan iteratorklassen og beholderklassen for dette ser ud. En skipliste kan principielt itereres på samme måde, men i stedet for at iteratoren kalder `get_prev` kaldes `get_backward`.



Figur 5.7: Baglænsnodeiterator- og beholderklassen.

5.1.3 Binær iterator

For at forklare en binær iterator (`binary iterator`) er det nødvendigt først at forklare hvad en sorteret beholder er, en sammenligningsfunktion (*comparotor*) er og hvad en sandhedsfunktion (*predicator*) er.

En sammenligningsfunktion gives to elementer og returnerer enten sandt eller falsk baseret på en sammenligning af de to elementer eller dele af elementerne.

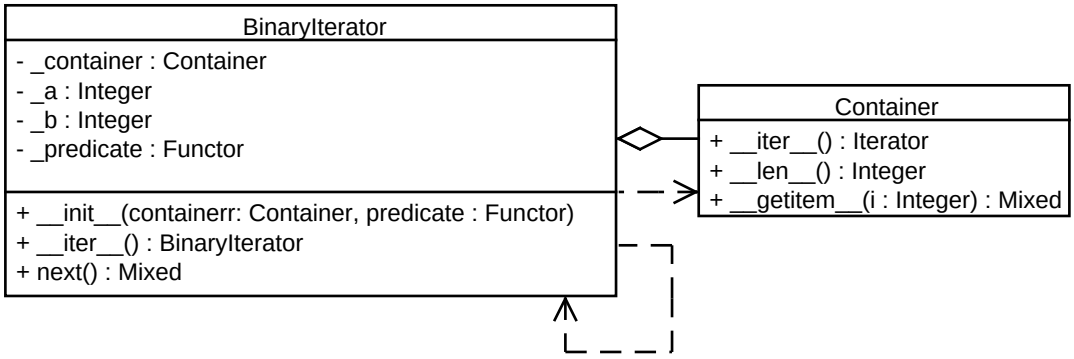
En beholder er sorteret, hvis det gælder for alle på hinanden følgende elementer, at de forårsager en sammenligningsfunktion, der ikke foretager en lig med eller forskellig fra test til at returnere sandt. Sorteringen er da den, der er implementeret af sammenligningsfunktionen.

En sandhedsfunktion gives et element og udfra test på dette element returnerer funktionen enten sandt eller falsk. Hvis en sandhedsfunktion enkapsulerer den sammenligningsfunktion, en beholder er sorteret efter og selv giver funktionen det manglende element, så vil den når den bliver kaldt med et element returnere sandt, hvis dette element er før i sortering end det den selv giver ellers returneres falsk.

En binæriterator itererer over en sorteret beholder ved hjælp af en sandhedsfunktion, der enkapsulere sammenligningsfunktionen beholderen er sorteret

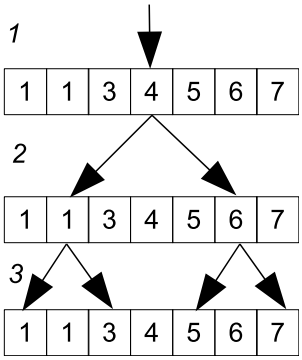
efter og som selv giver funktionen et mål element. Iteratoren gives en sekvens af elementer, for hver sekvens gives det midterste element til sandhedsfunktionen og hvis denne returnerer sandt sættes, sekvensen til at være fra midten til slut ellers fra start til midten. Den startende sekvens er alle elementerne i beholderen.

For en beholder der implementerer direkte indeksering (Python-list, vektor) og har en metode, der returnerer antallet af elementer i beholderen; kan en binær iterator implementeres ved, at den oprettes til at huske indeksene a (nul) og b (antallet af elementer i beholderen.), og for hver iteration, gives elementet i indeks $(b-a)/2$ til sandhedsfunktionen, hvis denne returnerer sandt sættes b til indekset ellers a og elementet og resultatet fra sandhedsfunktionen returneres. Hvis $b <= a$ rejses undtagelsen `StopIteration`. Figur 5.8 viser klassediagrammet for iterator- og beholderklassen og figur 5.9 (s. 75) viser de mulige stier, iteratoren kan tage afhængig af sandhedsfunktionen.



Figur 5.8: Binær indeksiterator og beholder klassediagrammet.

En binær iterator kan bruges til at søge efter et givent element i en sorteret beholder med få iterationer. Iteratoren kan også implementeres over linket lister, men her skal iteratoren iterere indtil midten af en sekvens for hver iteration, og dermed kan antallet af iterationer blive det samme som antallet af elementer i listen. Antallet af sammenligninger vil dog stadig være lavt. Hvis beholderen indeholder unikke elementer, kan iteratoren stoppes lige så snart det søgte element findes, men hvis beholderen indeholder ikke unikke elementer, og der søges efter det første af disse, skal den forsætte, indtil den rejser `StopIteration`. Figur 5.9 (s. 75) viser, at hvis der søges efter det første element 1, kan iteratoren ikke stoppe efter anden iteration, men først efter tredje.



Figur 5.9: Binær iteratorens mulige stier; pilene indikerer stierne og de kursive tal er iterationsnumrene.

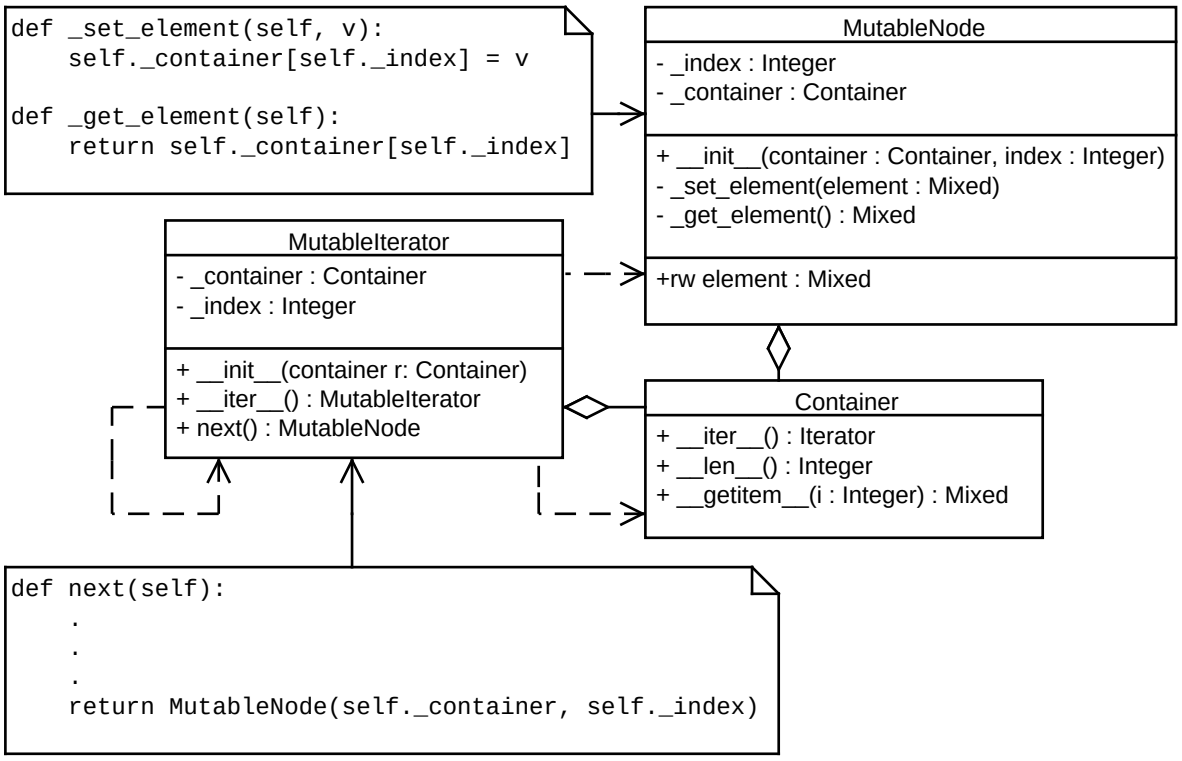
5.1.4 Foranderlige iteratorer

En foranderlig iterator (*mutable iterator*) er en iterator, der foranleder at elementerne i beholderen kan ændres ved at sætte værdien af elementerne i itereringen. Dette kan ikke udføres med Pythons iteratorkoncept, for eksempel ændre eksemplet i listing 5.4 ikke elementerne i beholderen.

Listing 5.4: Immutable iterering.

```
1 for element in container:
2     element = new_value
```

I Python er variabler reelt referencer til værdier og dermed er variablen *element* en reference til den værdi, som beholderen har en tilsvarende reference til, og i tildelingen af *new_value* til *element* sættes *element* til at refererer til den nye nye værdi, og dermed forbliver referencen i beholderen uændret. For at implementere en foranderlig iterator implementeres iteratoren så den returnerer en instans af en klasse `MutableNode`, som har metoder til at sætte og hente værdien af det element i beholderen, den er stedfortræder for. Figur 5.10 (s. 76) viser klassediagrammet for iterator-, beholder- og nodeklassen.



Figur 5.10: Klasserne for en foranderlig iterator baseret på en beholder der kan indekseres som et array.

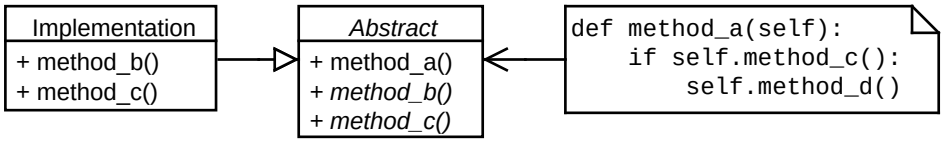
Med Python-egenskaben `element` i `MutableNode` klassen kan en iterativ tildeling af værdier til elementer i beholderen implementeres med følgende kodeeksempel.

```
1 for node in container:
2     node.element = new_value
```

At tildele en værdi til `element` i instansen af `MutableNode` kalder metoden `_set_element`, der sætter referencen i et indeks i beholderen til at referere til den nye værdi. Teknikken med at returnere en `MutableNode` instans eller en instans af tilsvarende klasse kan også bruges til iteratorer over for eksempel nodelister. (Det ville være en fejl blot at returnere noden i listen som en foranderlig node, da dette ville exponere nodernes link metoder til brugeren.)

5.2 Skabelonmetodemønstret

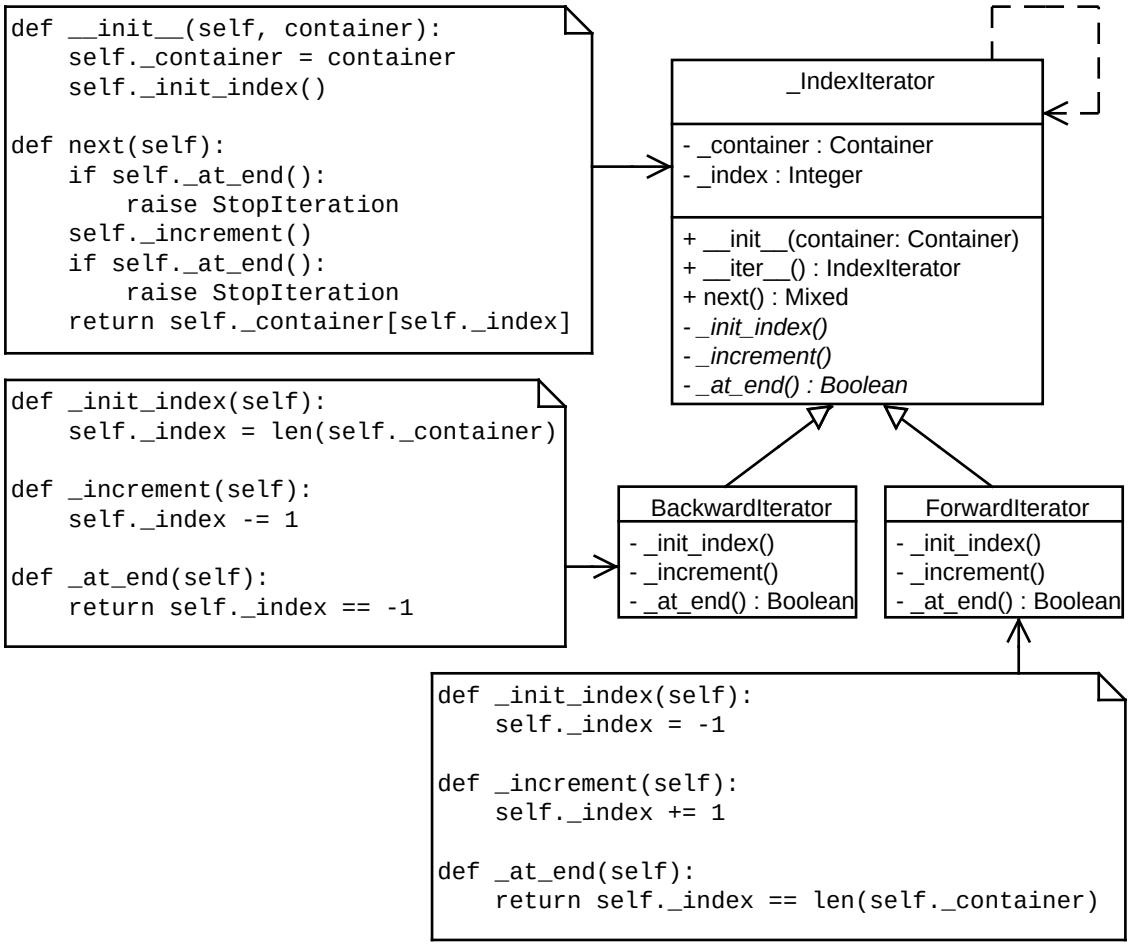
Denne sektion beskriver de konkrete anvendelser af skabelonmetodemønstret (*templatemethod pattern*) i CPH collections. Som nævnt i tabel 5.1 (s. 65) bygger mønstret på at implementere en konkret metode i en superklasse, hvis variable dele implementeres som kald til abstrakte metoder i klassen. De abstrakte metoder implementeres i underklasserne og dermed skal en underklasse kun implementere de dele af en metode, der er unik for klassen og lade superklassen implementere de konstante dele. Figur 5.11 (s. 77) viser klassediagrammet for skabelonmetodemønstret.



Figur 5.11: Klasserne i skabelonmetodemønstret.

5.2.1 Indeksiteratorer

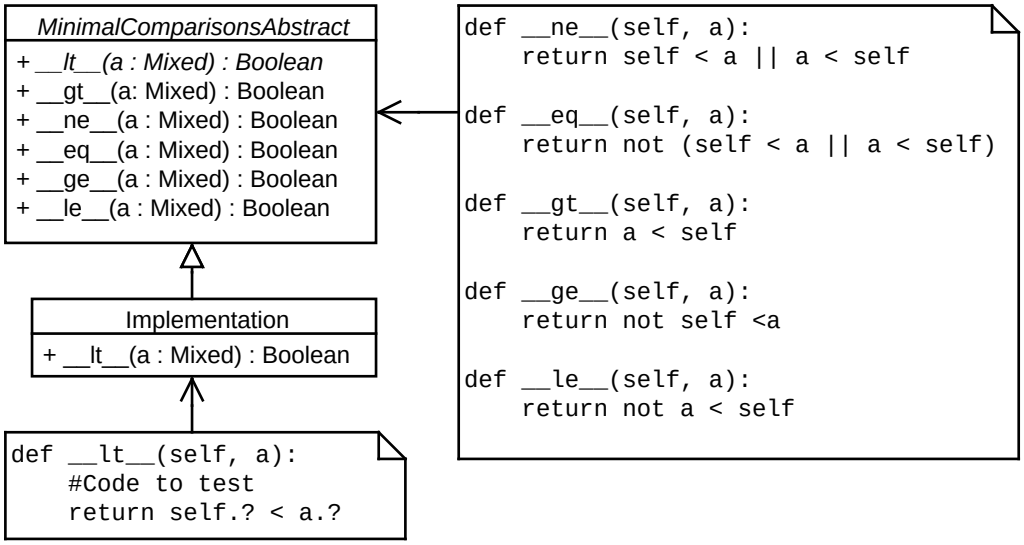
Iteratorerne for forlæns- og baglænsiterering baseret på indeksering er stor set ens. De starter med at sætte indekset til en værdi, og for hver iteration, sætter de indekset til en ny værdi, og tester om denne indikerer, at iterationen er slut, hvis den er rejses *StopIteration* ellers returneres elementet i det nye indeks. Derfor kan skabelonmetodemønstret bruges til at implementere disse iteratorer. Hver iteratorklasse skal blot have metoder til at sætte indekset i oprettelsen af iteratoren, opdatere indekset og fortælle om indekset indikerer slutningen af iterationen. Figur 5.12 (s. 78) viser klassediagrammet for indeksiteratorerne implementeret med skabelonmetodemønstret.



Figur 5.12: Klasserne i for indeksiteratorerne i skabelonindeksmønstret.

5.2.2 Minimalt antal sammenligningsmetoder

Minimalt antal sammenligningsmetoder (*Minimum comparison methods*) bruger skabelonmetodemønstret til at mindske antallet af sammenligningsmetoder, der skal implementeres i en klasse for at muligøre alle sammenligninger (==, ! =, <, <=, > og >=). Dette opnås ved, at superklassen, der implementerer mønstret, implementerer alle undtagen en af sammenligningsmetoderne (<). Disse metoder bliver implementeret som kald til den abstrakte metode, der ikke blev implementeteret i superklassen. Dermed skal klasser, der arver fra superklassen, kun implementere en sammenligningsmetode. Tabel 5.2 viser, hvordan de seks mulige sammenligningsrelationer alle kan udtrykkes med mindre end relationen og de boolske funktioner (`or` og `not`). Figur 5.13 viser klassediagrammet med implementeringen af metoderne i noter.



Figur 5.13: Klassediagram af mininum antal sammenligningsmetoder implementeringen.

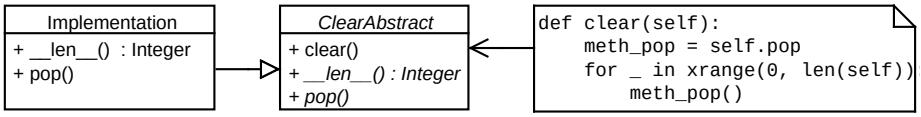
$a < b$	$a < b$
$a > b$	$b < a$
$a <= b$	$not\, b < a$
$a >= b$	$not\, a < b$
$a == b$	$not\, (a < b \,\|\|\, b < a)$
$a! = b$	$a < b \,\|\|\, b < a$

Tabel 5.2: Alle sammenligningsrelationer udtrykt kun med mindre end relationer.

Som det kan ses af figuren og tabellen er prisen for at bruge denne implementering, at der bliver foretaget ét ektra metodekald for sammenligningerne <=, > og >=. (En fortolker eller kompiler kan implementeres så den fjerner dette ekstra kald.) Prisen for at bruge implementeringen ved sammenligningerne == og ! = er dog to ekstra kald. Et til metoden selv, men også et ekstra kald til < metoden. (Som med de andre sammenligningsfunktioner kan kaldet til metoden selv undgås, hvis fortolkeren eller kompileren er implementeret til det. Det ekstra kald til < kan undgås, hvis fortolkeren eller kompileren er implementeret til at afbryde boolske eller (`or`) når den første del af udtrykket er sandt.) Denne brug af skabelonmønstret bygger endvidere på antagelsen om at de værdier, der sammenlignes er af samme type og derfor har samme metoder.

5.2.3 Tøm via pop

Hvis en beholdertype skal implementere en metode til at tømme beholderen (`clear`), så kan denne metode implementeres i en abstrakt superklasse som en løkke, der itererer antallet af elementer gange, og for hver iteration kalder beholderens "pop" metode. Pop-metoden er abstrakt i superklassen, da denne klasse ikke kan vide, hvordan en realisation af beholderen popper et element. Hvilken metode, der er pop-metoden afhænger af beholdertypen for eksempel er det `pop` i stak beholdere og `pop_back` i liste beholdere. Figur 5.14 viser klassediagrammet for denne implementering med Python-koden for `clear` skrevet i en note.



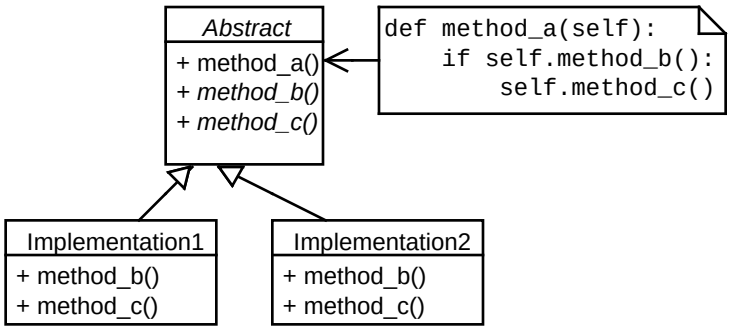
Figur 5.14: Klassediagrammet for tømning via pop.

Brugen af denne implementering betyder, at klasser der arver fra den abstrakte klasse ikke skal implementere `clear`-metoden, men eftersom implementeringen foretager et gennemløb over alle elementerne i beholderen og kalder "pop" hver gang, så kan en mere effektiv metode til tømning af en beholder formentlig implementeres i implementeringsklassen. Implementeringsklassen ved mere om datastrukturen, der gemmer elementerne for beholderen. Vektorer kan ikke bruge denne teknik, da tømning via `pop_back` formentlig vil medføre optil flere reallokeringer af elementerne i en vektor under tømningen.

5.3 Abstrakt superklassegrænseflademønstret

Denne sektion beskriver de konkrete anvendelser af abstrakt superklassegrænseflademønstret i CPH collections. I CPH collections bruges dette mønster til at definere grænsefladen, som implementeringer af de forskellige beholdertyper skal implementere.

Fordelen ved dette mønster er, at algoritmer, der kræver en bestemt beholdertype eller er implementeret forskelligt afhængig af beholdertypen, kan teste om en given variabel er en instans af den abstrakte superklasse for den pågældende beholdertype. Hvis dette mønster ikke bruges, skal algoritmerne teste, om variablen er en instans af en af implementeringerne af en beholdertype eller om instansen har grænsefladen for en beholdertype. Figur 5.15 (s. 81) viser klassediagrammet for mønstret.



Figur 5.15: Klasserne i abstrakt superklassegrænseflademønstret.

Hvis algoritmerne tester, om en instans er af en specifik beholdertype, ville de kræve ændringer for hver ny implementering af en beholdertype, og hvis de tester på, om en instans har en grænseflade kunne de få falsk positiver på instanser, der har grænsefladen, men som ikke er af den beholdertype.

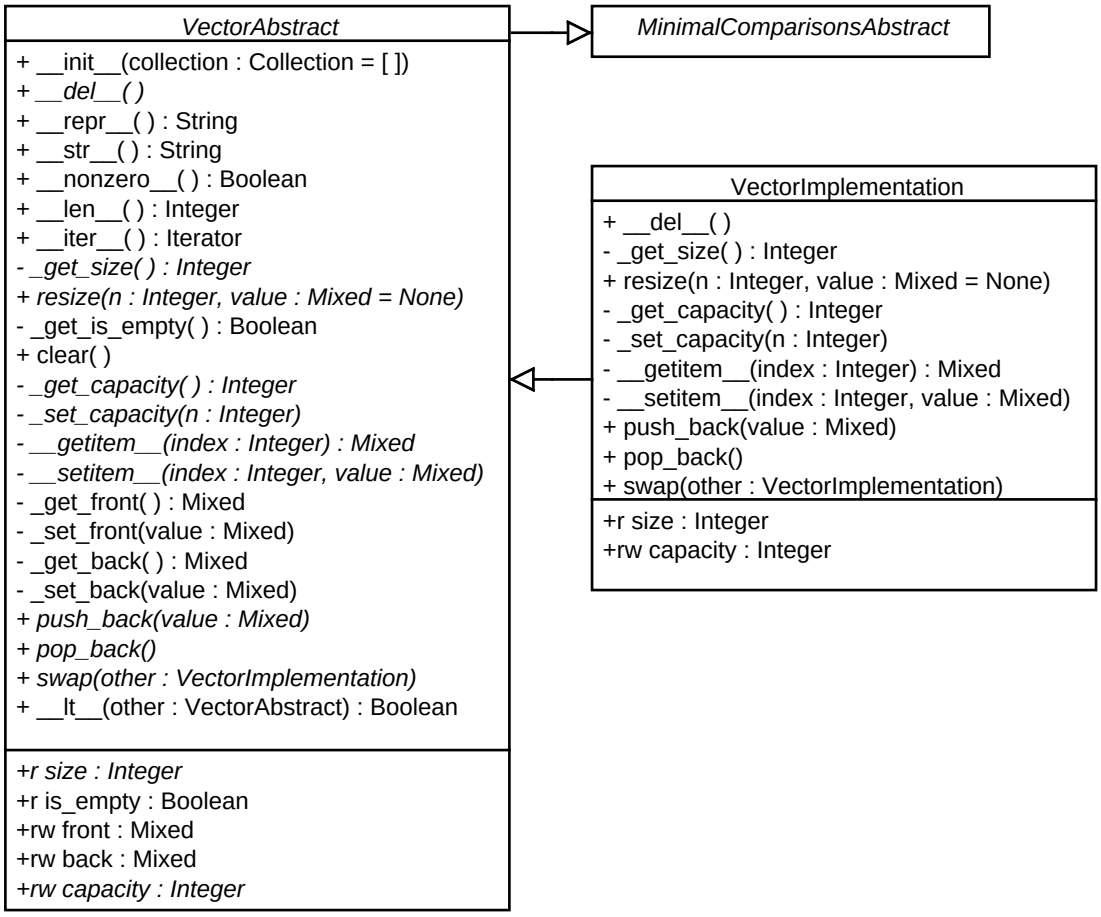
Mønstret kan blandes sammen med at implementere nogle af beholdertypens metoder i den abstrakte klasse, disse metoder vil da bruge kald til abstrakte metoder i klassen. (De implementerede metoder i den abstrakte klasse gør dermed brug af Skabelonmetodemønstret.), se `method_a` i figur 5.15.

Dette mønster garanterer kun, at en klasse, der arver fra den abstrakte klasse implementerer grænsefladen[3] Mønstret garanterer dog ikke, at en implementering overholder beskrivelsen af en beholdertype. Det er muligt at implementere en klasse, der opfylder en grænseflade, men som udfører noget helt andet end det som forventes af instanser af beholdertypen.

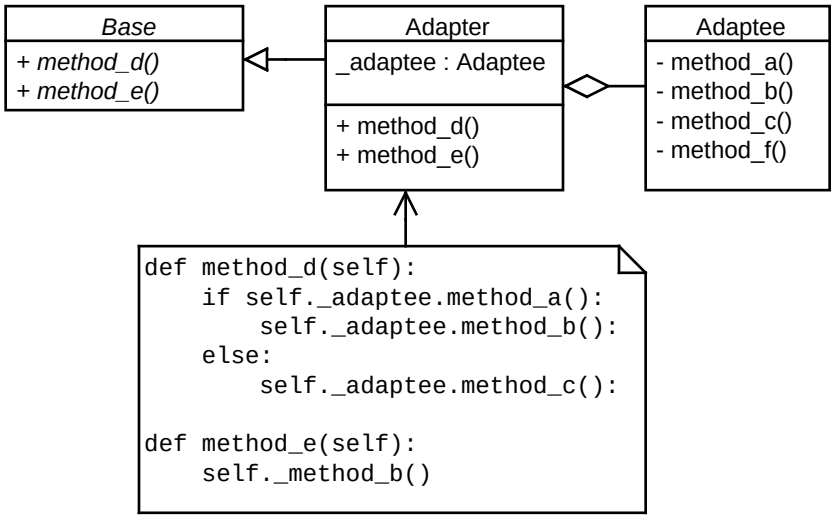
Alle beholdertyperne vektor, list, stak, kø, toendetkø, kort, multikort, multisæt, sæt og prioritetskø har i CPH collections hver deres abstrakte superklasse. Fælles for klasserne er, at de arver fra den abstrakte klasse `MinimumCompareAbstract` se sektion 5.2.2 (s. 79), og at de selv implementerer den manglende mindre end relation. Ligeledes implementerer klasserne også en del andre metoder, som enten kan implementeres som kald til abstrakte metoder i klassen eller som kald til implementerede algoritmer.

[3] Det er ikke muligt at oprette instanser af en klasse, der mangler implementeringer af abstrakte elementer.

Dermed er en klasse, der arver fra dens respektive superklasse ikke alene garanteret, at den implementerer grænsefladen for den givne beholdertype, men antallet af metoder, som skal implementeres, er også mindre. Figur 5.16 (s. 82) viser den abstrakte superklasse for vektorer, de abstrakte superklasser for de andre beholdertyper kan findes i bilag D (s. 149).



Figur 5.16: Klassediagrammet for den abstraktebaseklasse for vektor implementeringer.



Figur 5.17: Klasserne i adaptermønstret.

5.4 Adaptermønstret

Denne sektion beskriver de konkrete anvendelser af adaptermønstret (*Adapter pattern*) i CPH collections. Som nævnt i tabel 5.1 (s. 65) bygger dette mønster på at implementere en adapter klasse, der opfylder en grænseflade, men som blot rediregerer kald til denne grænseflade til en underliggende datastruktur (adapterede), som ikke opfylder grænsefladen. Det er ikke et krav, at adapterklassen kun har en instans af denne datastruktur. Adapter klassen kan bruge yderligere dataer til at implementere manglende funktionaliteter i den adapterede datastruktur. Dette mønster kan implementeres som en aggregering og eller via arvning.

Forskellen er, at ved aggregeringen kan den adapterede datastruktur være alle datastrukturer, der arver fra den specificerede datastruktur[4,5], og ved privat (*private*) arvning[6] er det kun den specifikke datastruktur der adapteres. Figur 5.17 viser det generelle klassediagram for adaptermønstret, når der bruges aggregering.

Som det kan ses på figuren (noten), så behøver adapterklassen ikke at kalde alle metoderne (`metode_f`) som er i den adapterede datastruktur, og den behøver heller ikke at foretage direkte kald til metoder i den adapterede datastruktur, men kan kombinere flere kald til datastrukturen (`metode_d`).

[4] I Python bliver typen på variabler ikke specificeret og derfor kan det være alle datatyper, der har den grænseflade som bruges på den aggregerede datastruktur.

[5] Forudsætter at det er muligt at sætte typen eller realisationen der adapteres.

[6] Python har kun koncepter som privat, offentlig og beskyttet via navngivning og derfor kan en arvnings aggregering ikke implementeres.

Risikoen ved brug af adaptermønstret er, at datastrukturen, der adapteres, ændres. Ændringer i denne struktur kan medføre, at adapterklassen bliver ukorrekt og dermed forårsager fejl i fortolkningen eller kompileringen. Ændringer kan også forårsage fejl, der først opdages under kørsel. Disse fejl kunne for eksempel være ændringer af retur værdier eller hvilke undtagelser datastrukturen rejser.

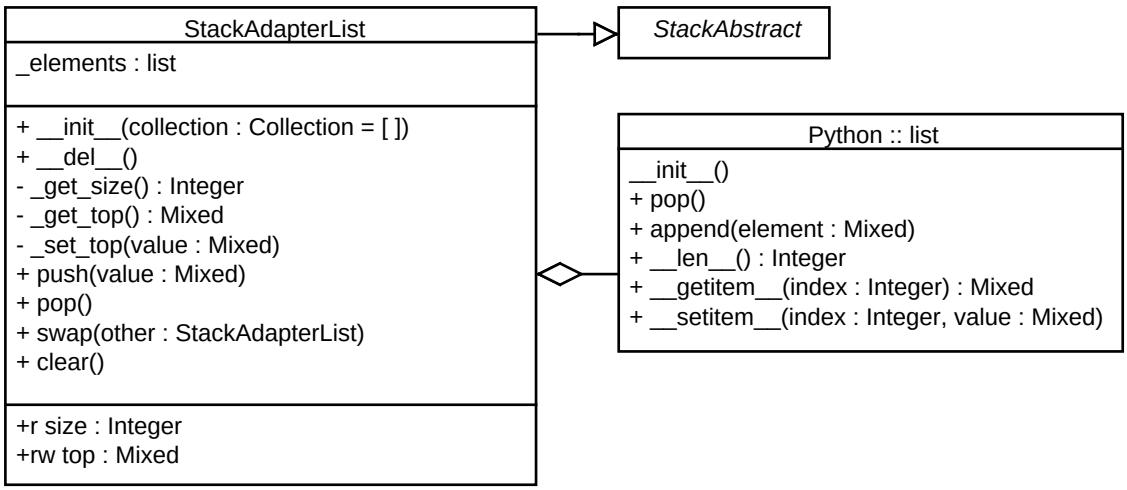
5.4.1 Stak implementeret som en adapter af Python-list

En mulig implementering af en stak er at bruge adaptermønstret til at adaptere fra Pythons indbyggede listetype.

Denne adaptering forårsager dog, at operationerne `push` og `pop` ikke kan garanteres at være i konstant tid, da de kalder listens `append` og `pop`, der begge kan forårsage en reallokering af elementerne i listen.

I denne adaptering er det øverste element i stakken det sidste element i listen så, for at foretage en forlænsiteration af stakken, skal der bruges en baglænsiteration af listen, og for at foretage en baglænsiteration, skal der bruges en forlænsiteration af listen.

Ligeledes, eftersom at dette er en adaptering af Pythons liste, er hukommelsen stakken bruger ikke linæer i antallet af elementer, som stakken indeholder. Figur 5.18 viser klassediagrammet for denne implementering, dog er kun metoder fra list klassen, som adapteren bruger, vist.



Figur 5.18: Klassediagrammet for stak adaptationen af list.

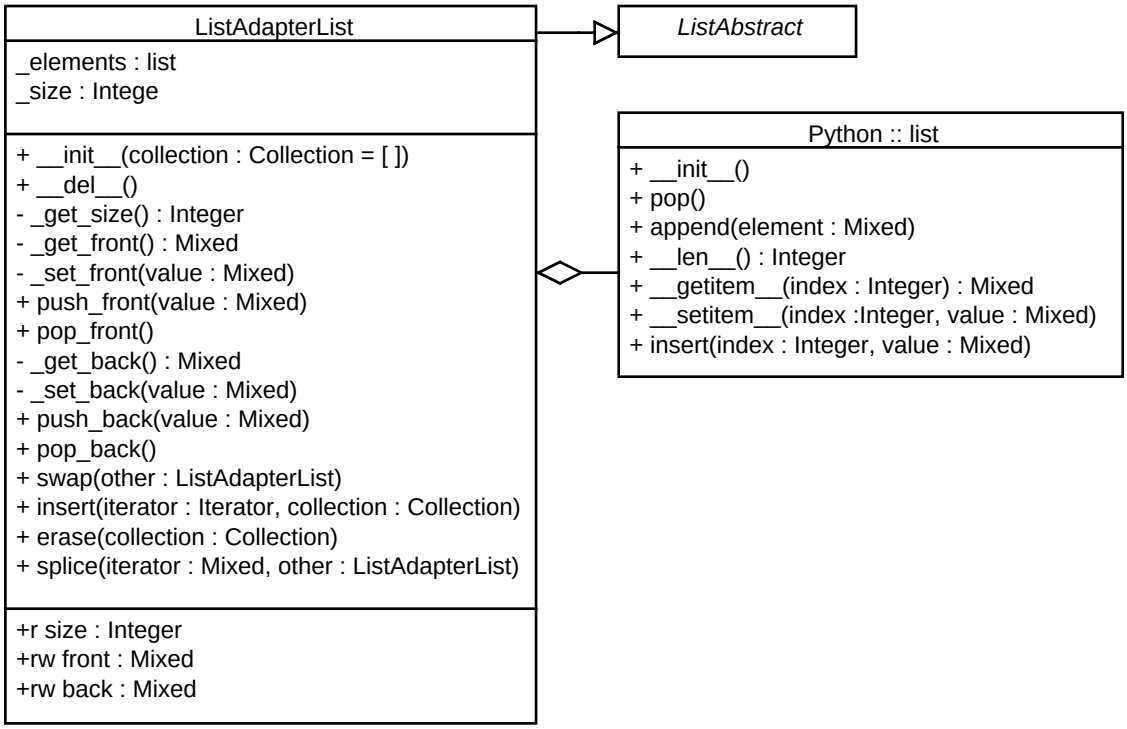
5.4.2 List implementeret som en adapter af Python-list

En mulig implementering af en liste er at bruge adaptermønstret til at adaptere fra Pythons indbyggede listetype.

Denne adaptering forårsager dog, at metoder, der ændrer i antallet af elementer i listen, ikke er konstante, men kan være linæere i antallet af elementer. Dette gælder især metoderne, der ændrer i starten eller midten af listen.

Elementerne i listen er i samme rækkefølge som i Pythons liste, og derfor kan index iteratorerne bruges til at iterere over listens elementer.

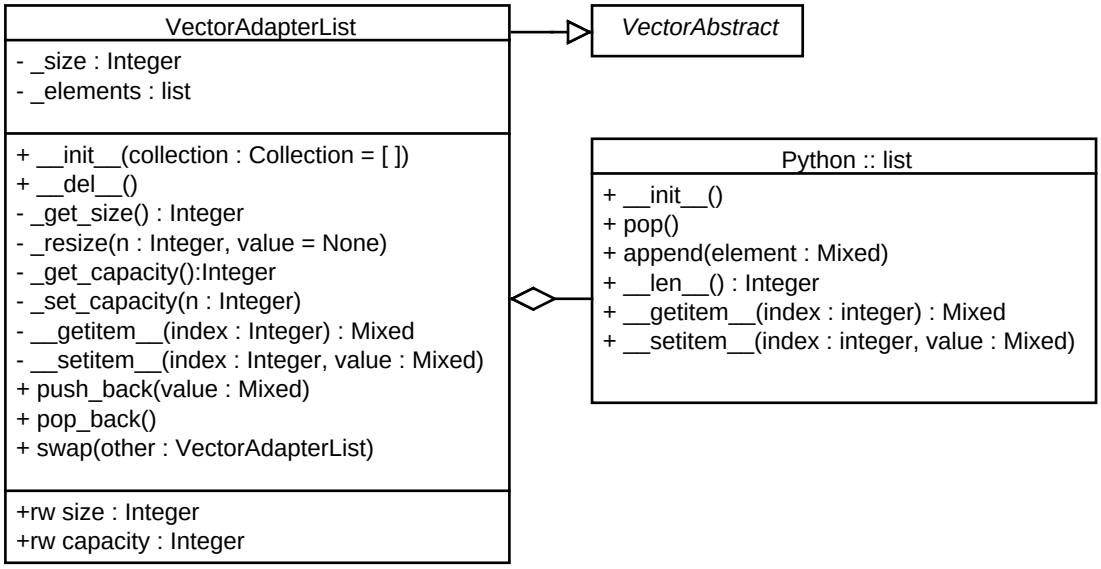
Endvidere er hukommelsen som listen bruger ikke linæer i antallet af elementer den indeholder. Figur 5.19 viser klassediagrammet for denne implementering, men dog er kun metoder fra list klassen, som adapteren bruger, vist.



Figur 5.19: Klassediagrammet for liste adaptationen af Python list.

5.4.3 Vektor implementeret som en adapter af Python-list

En vektor kan implementeres ved brug af adaptersmønstret til at adaptere fra Pythons indbyggede listetype.



Figur 5.20: Klassediagrammet for vektor adaptationen af Python list.

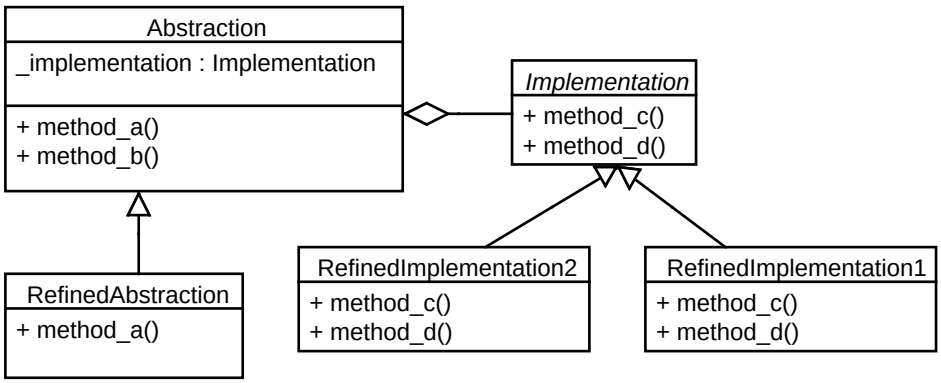
Denne adaptering bruger antallet af elementer i listen som kapaciteten i vektoren og implementerer selv en værdi for vektorens størrelse (antal af elementer). Pythons liste implementering har selv en kapacitet, men denne er utilgængelig fra dens grænseflade, og derfor har vektoren kapacitet til mere end den kapacitet den er sat til[7]. Figur 5.20 (s. 86) viser klassediagrammet for denne implementering, men dog er kun metoder fra list klassen, som adapteren bruger, vist.

5.5 Bromønstret

Denne sektion beskriver de konkrete anvendelser af bromønstret i CPH collections. Som nævnt i tabel 5.1 (s. 65) bygger dette mønster på at implementere klasser af to grænseflader (abstraktion og implementering). Klasser af abstraktionsgrænsefladen er dem, som brugeren får instanser af, og disse klasser bruger instanser af klasser af implementerings grænsefladen. Dette betyder, at abstraktion og implementering kan varieres uafhængig af hinanden.

Afhængig af broens implementering vælges realisationen enten af broen selv eller gives til broen. (Den sidste mulighed kræver, at broen accepterer en reali-

[7] Listens kapacitet er større end antallet af elementer i den og vektorens kapacitet er antallet af elementer i listen.



Figur 5.21: Klasserne i bromønstret.

sation i sin grænseflade, og er dermed en udvidelse typernes eksisterende grænseflade.) Figur 5.21 (s. 87) viser det generelle klassediagram for bromønstret.

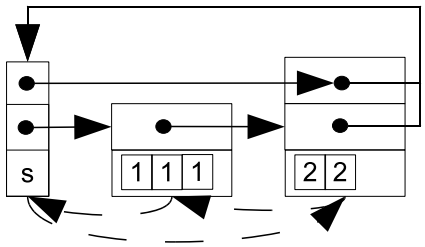
I CPH collections findes der implementeringer af en bro til vektor, toendetkø, sæt, multisæt, kort, multikort og prioritetskø. Figur 5.23 (s. 89) viser broen for et sæt. Ifølge figuren skal implementeringen implementere metoderne `lower_bound` og `upper_bound`, dette er teknisk set ikke korrekt. Er metoderne ikke til stede i implementeringen gennemløber sæt abstraktionen elementerne i implementeringen via iteratorer i de tilsvarende metoder.

Kravene til implementerigens grænseflade er den samme for alle broerne undtagen vektorbroen og toendetkøbroen, og derfor kan de samme implementeringer bruges i de andre broer.

Multisæt og -kort broerne gemmer ikke elementer enkeltvis i implementeringer, men derimod gemmes de Python-lister med ens elementer i implementeringen og dekorerer udtræksfunktionen til at trække nøglen ud af det første element i disse lister, se figur 5.22 (s. 88). Dette betyder, at antallet af operationer for at finde et element i sæt- og multisætbroen er de samme, hvis implementeringsklassen har metoderne `lower_bound` og `upper_bound`. Men det betyder også at de to broer har deres egne iteratorer; der kombinerer implementeringens iterator og med en iterator over lister.

Kravene til implementeringsgrænsefladen for sæt, multisæt, kort, multikort og prioritetskø betyder, at implementeringerne skipliste, se sektion 4.1.2 (s. 55), og binært søgetræ, se sektion 4.1.3 (s. 58), kan bruges som implementering i broerne. Som det kan ses i klassediagrammet i figur 5.23 (s. 89) er der ingen nedarvning, men som nævnt i starten af dette kapitel er dette ikke nødvendigt i Python for implenteringernes vedkommende og for abstraktionens vedkommende kan der altid implementeres en klasse, der arver fra sætbroen. Klassediagrammerne for de andre broer kan ses i bilag E (s. 160).

Broen for den toendetkø er en bro til implementeringer, der har listegrænsefladen. Som nævnt tidligere er den eneste forskel i disse to grænseflader mulig-



Figur 5.22: Skiplisten brugt i et multisæt til at holde tal.

heden for at fjerne og indsætte elementer andre steder end i enderne for listegrænsefladens vedkommende. Broen for vektoren er en bro til en implementering, der har en mindre grænseflade end vektoren selv og som antager, at indeks givet til den er tjekket, før de gives.

Figur 5.23: Klasserne i sæt-bromønstret.

5.6 Klassefabrikmmøstret

Denne sektion beskriver de konkrete anvendelser af Klassefabrikmønstret i CPH collections. I CPH collections bruges mønstret til at lade brugerne abstrahere væk fra navnene på realisationerne af de enkelte beholdertyper og i stedet bruge andre parametre til at vælge en realisation.

Dette mønster svare ikke til "Abstract Factory" [6]. Forskellen er, at i "Abstract Factory" vælger brugeren mellem konkrete fabrikker, mens brugeren i dette mønster bruger fabrikken til at vælge mellem konkrete realisationer. Mønstret svarer heller ikke til "Factory Method" [6], hvor konstruktionen af de konkrete typer implementeres i en fabriksmetode i en superklasse i arvehierakiet.

Ideen til denne opbygning er baseret på beskrivelsen af *lookup* i [15](kap. 7).

5.6.1 Beholdertypefabrikker

I CPH collections er der implementeret en klassefabrik for alle de beholdertyper, som en bruger kan få instanser af. Disse typer er vektor, liste, kø, toendetkø, stak, sæt, multisæt, kort, multikort og prioritetskø.

Fælles for alle fabrikkerne er, at de har samme navn, som den type de er fabrik for, og, at de nedarver fra en fælles fabrik. (Fabrikken for stakke kaldes *Stack*.). Nedarvningen mindsker mængden af kode nødvendig for individuelle fabrikker, og med navngivningen opnås, at brugeren kan bruge beholdertypens navn til at få en realisation af den pågældende beholder.

Konstruktøren i den fælles fabrik vælger en specifik realisation, dette foretages udfra nøgleordsargumenter givet til fabrikken. Fabrikken forstår kun nøgleordsargumenterne `impl` og `deco`. Nøgleordsargumenter, der bruges af en fabrik, bliver ikke givet som argumenter til realisationen, men alle andre værdier gør. Ideen til at designe en fabrik på denne måde er fra [1].

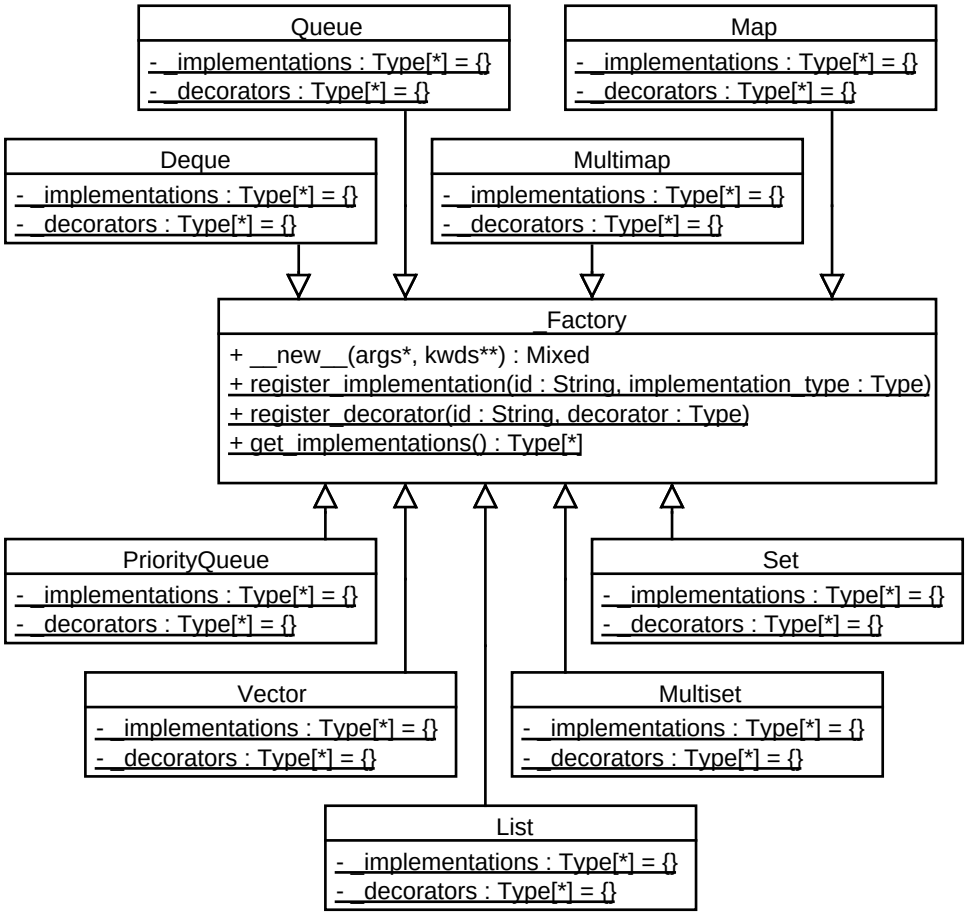
Hver realisation kan fås, ved at give `impl` som teksten bestående af en realisations store bogstaver skrevet som små bogstaver. (`StackAdapterList` ville kunne fås med `impl = ''sal''`.). Dette betyder, at brugeren skal kende navnet på den enkelte realisation, men fabrikken kunne udvides til at forstå nøgleord, der i stedet beskriver opførelsen af en realisation, som for eksempel `speed` eller `memory`.

Realisationerne, en fabrik kender, er afhængig af hvilken type den er fabrik for, så det er ikke muligt at få en realisation af en vektor gennem fabrikken for stakke.

Fabrikken kan sætte dekoratorer om en realisation. Dette gøres, ved at give fabriken en liste af dekoratorernes navne som nøgleordsargumentet `deco`. Hver dekorator skal selv registrere sit navn i den pågældende fabrik. Dekoratorer gives en realisation efter at realisationen er blevet oprettet. (En realisation kan være en dekorator, hvis flere dekoratorer er givet i listen.) Lige som med realisationer registreres dekoratorer for de individuelle fabrikker, så det er ikke muligt at få en dekorator til vektortypen til staktypen.

Figur 5.24 (s. 91) viser klassediagrammet for beholderfabrikkerne og i tabel 5.3 (s. 92) kan læses hvilke identifikationstekster de individuelle realisationer og

dekoratorer er registreret med i deres respektive fabrikker.



Figur 5.24: Klassediagrammet for klassefabrikkerne; læg mærke til, at alle fabrikkerne har variablerne `_decorators` og `_implementations`. Grunden til dette er, at eftersom de er klassevariabler, kan de ikke lægges i klassen `_Factory`, da de så ville gælde for alle fabrikkerne.

ID	Klassenavn
	Vektor - realisationer
vb	_VectorBrigde
val	_VectorAdapterList
	Vektor - dekoratorer
copy	_CopyDecoratorVector
homogen	_HomogenDecoratorVector
index	_IndexDecoratorVector
	Liste - realisationer
lal	_ListAdapterList
ll	_ListLinkedlist
	Stak - realisationer
sc	_StackC
sl	_StackLinkedlist
sal	_StackAdapterList
	Kø - realisationer
ql	_QueueLinkedlist
	Toendetkø - realisationer
dl	_DequeLinkedlist
dbl	_DequeBridgeList
	Prioritettskø - realisationer
pqb	_PriorityQueueBridge
	Kort - realisationer
mb	_MapBridge
	Multikort - realisationer
mb	_MultimapBridge
	Sæt - realisationer
sb	_SetBridge
	Sæt - dekoratorer
copy	_CopyDecoratorSet
	Multisæt - realisationer
mb	_MultisætBridge

Tabel 5.3: Viser identifikationstekten en klasse er registreret med i en fabrik.

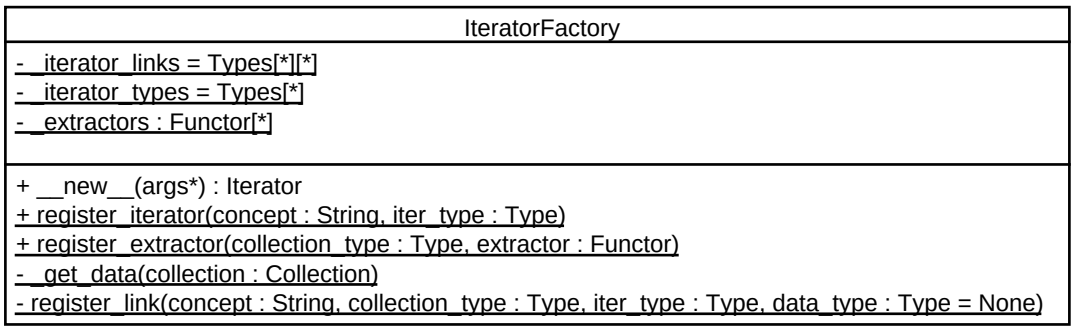
5.6.2 Iteratorfabrikken

I CPH collections er iteratorer adskilt fra deres beholdere grundet et designvalg. Dette ville i praksis betyde, at en bruger skulle vide specifikt hvilken iterator, der passede til en beholdertype. For at undgå dette er der i CPH collections en iteratorfabrik.

Iteratortyper skal registreres i denne fabrik under et iteratorkoncept (forlæns, baglæns, etc..) og beholderimplementeringer kan registrere udtræksfunktioner til at udtrække den iterable data, og de kan registrere en binding til en specifik iteratortype for et erklæret iteratorkoncept.

Det er nødvendigt for en beholder at registrere en udtræksfunktion da, der i beholdernes grænseflader (liste, stak, etc.) ikke er nogen metoder til at få den iterable data. Det er nødvendigt at kunne binde til en specifik iteratortype for en beholder, hvis den iterable data er bundet til en anden iteratortype. For eksempel, som nævnt tidligere, findes der en implementering af stak, som adaptere fra en liste, men iteratoren til stakken er omvendt af listen, og derfor skal denne implementering registrerer listens baglænsiterator som sin forlænsiterator i fabrikken. Figur 5.25 viser klassediagrammet for iteratorfabrikken.

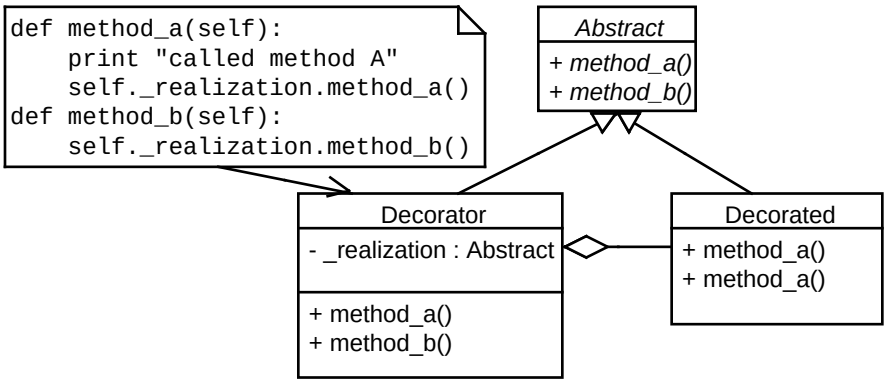
Når der skal bruges en iterator for en samling, gives samlingen samt et iteratorkoncepts navn (tekststrenge som for eksempel `"forward"`.) til fabrikken. Fabrikke slår samlingstypen op i en tabel for at se, om en iteratortype til samlingen med det ønskede koncept er registreret og ellers gennemløbes alle registrerede iteratorer af det ønskede koncept, og hver iteratortypes klassemetode `accept` kaldes med samlingen. Metoden `accept` returnerer sandt, hvis den kan foretage en iterering af den givne type.



Figur 5.25: Klassediagrammet for iteratorfabrikken.

5.7 Dekoratormønstret

Denne sektion beskriver de konkrete anvendelser af dekoratormønstret (*Decorator pattern*) i CPH collections. Som nævnt i tabel 5.1 (s. 65) bygger dette mønster på at implementere en konkret klasse, der opfylder en grænseflade, men som udlukkende udvider eksisterende implementeringer af samme grænseflade med ny funktionalitetet. En dekorator klasse er dermed ikke i sig selv funktionel, uden den har en instans af den type, som den dekorerer. Dette mønster implementeres som en aggregering og figur 5.26 viser det generelle klassediagram for dekoratormønstret.



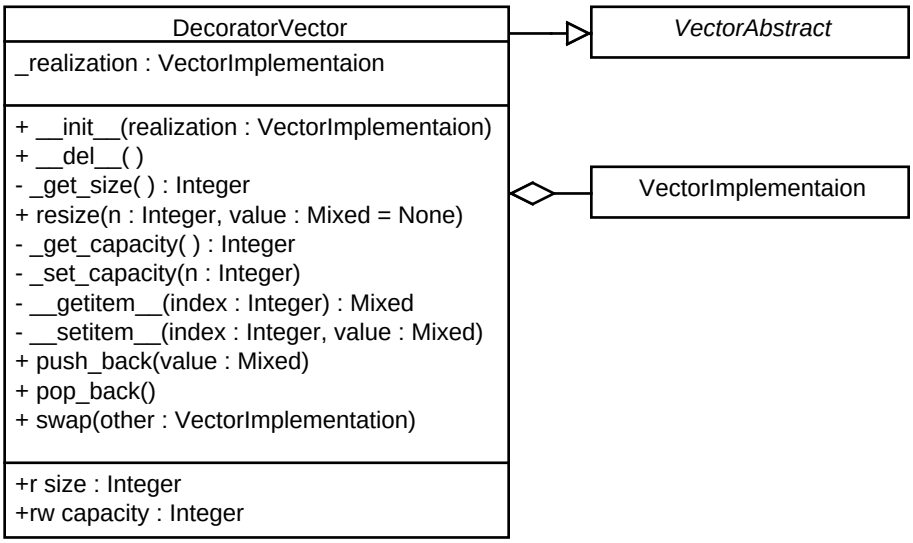
Figur 5.26: Klasserne i dekoratormønstret.

I CPH collections forventes det, at dekoratoren accepterer en instans af en realisation i oprettelsen og ikke dens type. Effekten af dette er, at realisationen allerede skal være oprettet, når dekoratoren bliver lagt om den. Eftersom, at realisationen allerede er oprettet, når dekoratoren får den, må dekoratoren efterbehandle en realisation for at sikre, at denne opfylder dekoratorens funktionalitet.

Grunden til at dekoratorer implementeres til at acceptere en instans af en realisation er, at det på det givne tidspunkt var den indlysende løsning for at gøre det muligt at lægge mere end en dekorator om en realisation. Eftersom at dekoratorer kun bruger grænsefladen de dekorerer kan de tage en instans af dekorator som realisation, forudsat at det før eller senere er en dekorator med en rigtig realisation.

En dekorator fungerer som en ektra skald omkring en realisation (eller dekorator) og derfor tillægger dekoratoren en ektra omkostning ved alle kald til metoder, selvom den pågældende metode ikke har noget med funktionaliteten som dekoratoren implementerer. (Kaldet går først til dekoratoren og fra dekoratorens metode til realisationens.) Et alternativ til dekoratormønstret er strategimønstret.

Stategimønstret giver en funktionalitet til realisationen i form af et objekt eller funktion, som realisationen kan bruge i de metoder funktionaliten skal



Figur 5.27: Klassediagrammet for vektorens basisdekorator.

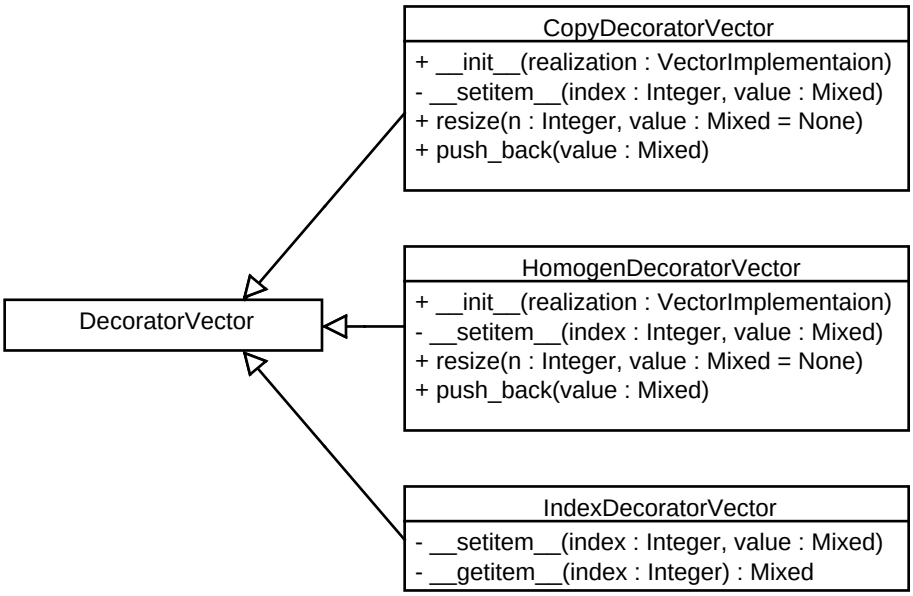
bruges. Fordelene ved strategimønstret er dels, at en strategi kan implementere funktionalitet i metoder, der ikke er en del af grænsefladen, og at den nye funktionalitet kun koster i de metoder, der anvender den. Ulemperne ved strategimønstret er, at der skal ændres i en realisations kode og grænseflade for henholdsvis at bruge og accepterer strategier. (Hvis alle implementeringer skal kunne bruge en strategi skal de alle ændres.), og at uanset om en funktionalitet ønskes brugt eller ej, så bruger realisationenerne altid resourcer på at finde ud af om dette er tilfældet i metoden[8].

Eftersom, at det ikke er alle metoder i en dekorator, der bruges for at implementere en funktionalitet, kan der implementeres en basisdekorator, der implementerer alle metoder i grænsefladen som kald til deres tilsvarende metoder i realisationen. Reelle dekoratorer kan arve fra basisdekoratoren og kun implementere de metoder, der skal bruges for at implementere deres funktionalitet.

Figur 5.27 viser klassediagrammet for vektorens basisdekorator og figur 5.28 (s. 96) viser klassediagrammet for de tre eksisterende vektordekoratorer i CPH collections, se tabel 5.3 (s. 92). Som det ses af figuren for basiedekoratoren af vektoren arver denne fra vektoren abstrakte superklasse og kan dermed bruges i algoritmer der tester for denne arvning.

CopyDecoratorVector: Er kopieringsdekoratoren til vektorer. I C++ `STL` tages en kopi af værdier, når de gives til vektoren og det er kopierne, der

[8] Det er ikke nødvendigvis sandt i Python, hvor en metode kan sættes til at pege på en anden implementering.



Figur 5.28: Klassediagrammet for vektordekoratorer.

lægges i vektoren. I implementeringer af vektorer i CPH collections lægges referencen, som gives, i vektoren. Dette er væsenligt hurtigere end at kopiere værdien i Python og vil i de fleste tilfælde ikke give problemer. Men for at kunne opfylde C++0x specifikationen [2] er der implementeret en kopieringsdekorator til vektorgrænsefladen.

Dekoratoren arver fra basisdekoratoren og behøver dermed kun at implementere få af metoderne i vektorgrænsefladen, og eftersom at foranderlig iteratoren for vektoren bruger vektorens grænseflade, så vil iteratoren også kunne bruge dekoratoren, og dermed bliver dekoratorens funktionalitet indirekte ført ind i iteratoren.

At en dekorering af beholderensgrænsflade medføre en dekorering af iteratoren gælder ikke for alle beholdertyper. For eksempel ville det ikke gælde for en dekorator af listegrænsefladen eftersom listeiteratorer ikke bruger grænsefladen til at iterere. Derfor ville en kopierings dekorator af denne grænseflade også skulle have en dekorator for foranderlighedsiteratoren for typen.

HomogenDecoratorVector: Er homogendekoratoren til vektorer. I C++ `STL` kan der kun lægges en type (eller undertyper) af elementer i en instans af en vektor, men i CPH collections er det muligt at lægge en hvilken som helst type i en vektor. Vektorer i CPH collections er dermed heterogene, mens de er homogene i C++ `STL`.

For at gøre vektorer homogene er implementeret en dekorator. Denne tager typen af det første element i vektoren som den eneste type af elementer, der må lægges i vektoren. (Hvis vektor realisationen ikke er tom, er dette indeks 0.)

Som ved kopieringsdekoratoren fungerer dekoratoren med foranderlighedsiteratoren.

Implementeringen af homogenitet er forskellig fra den i C++ `STL`, hvor typen gives til vektoren og vektoren ikke selv udleder den. Denne forskel udmønter sig i, at C++ `STL` kan gives en type højere op i arvehierakiet end de elementer, der lægges i vektoren og dermed kunne tage alle undertyper af denne, mens det i dekoratoren er typen af det første element, der bliver den tilladte type og alle dens undertyper.

IndexDecoratorVector: Er indeksdekoratoren til vektorer. I C++ `STL` tester vektorer ikke om det indeks, der gives i direkte indeksering, faktisk er et lovligt indeks. Dette testes heller ikke i vektorer i CPH collections, men da det kan tænkes, at nogen ønsker, at dette bliver gjort, er der denne dekorator. Som ved kopieringsdekoratoren fungerer dekoratoren med foranderlighedsiteratoren.

I CPH collections findes endvidere implementeringer af en basis- og kopieringsdekorator for sæt.

Kapitel 6

Algoritme implementeringsmønster

Dette kapitel beskriver det generelle mønster algoritmer implementeret i CPH collections følger. I CPH collections er nogle af algoritmerne fra C++0x specifikationen [2] implementerede, men ikke dem alle. Tabel 6.1 (s. 99) lister de implementerede algoritmer.

Når algoritmerne implementeres, så de følger designvalgene beskrevet i kapitel 2 (s. 4) bliver dele af deres implementeringskode ens, og det er dette mønster, der bliver beskrevet. Beskrivelsen vil tage udgangspunkt i implementeringen af algoritmen *seach* fra biblioteket, koden til denne er vist i listing 6.1 (s. 99), når der refereres til linie numre i den efterfølgende liste er det til koden for algoritmen.

Linie 1-4 **og** 11 - 61**:** Er dokumentation skrevet i det valgte dokumentations format som blev valgt i designvalget beskrevet i sektion 2.1.11 (s. 12).

Linie 5**:** Det er nok kun at importere bibliotekets hovedpakke for at have tilgang til det hele. Dette er en bieffekt af designvalget i sektion 2.1.7 (s. 9), om at fabrikkerne på første niveau i biblioteket, og fra designvalget i sektion 2.1.5 (s. 8), om at pakker importerer hele deres indhold.

Linie 6-7**:** Der skal være to blanke linier før en funktions erklæring i den valgte kodestil fra designvalget i sektion 2.1.10 (s. 10).

Linie 8-10**:** Funktionens definering er linie brudt efter hvert komma for at komme under den tilladte linie længde i den valgte kodestil fra designvalget i sektion 2.1.10 (s. 10). Samtidig kan ses at funktionen følger navngivnings princippet for argumenterne i den valgte kodestil.

Linie 62**:** Algoritmen skal bruge en iterator til den anden samling, som algoritmen gives. Hvis samlingen selv er en iterator, tages der en kopi af denne. Dette skyldes designvalget i sektion 2.1.9 (s. 10), om at implementere algoritmer til at tage samlinger i stedet for iteratorer.

adjacent_find	append	assign
binary_search	clear	count
count_if	equal	extend
find_if	find	includes
index	lexicographical_compare	lower_bound
make_heap	merge	mismatch
partition	pop_heap	push_heap
random_shuffle	remove	remove_if
reverse	reverse_copy	rotate
rotate_copy	search	search_n
set_difference	set_intersection	set_symmetric_difference
set_union	sort	sort_heap
stable_partition	stable_sort	swap
unique	upper_bound	

Tabel 6.1: Alfabetisk listning af implementerede algoritmer i CPH collections.

Linie 63 - 66**:** Verificerer om funktionen fik en iterator til den anden samling, og hvis ikke hentes en undtagelse fra den globale undtagelsesstyring fra designvalget i sektion 2.1.8 (s. 9). Den valgte kodestils linie længde følges ved at liniebryde ved hvert komma i argumentlisten.

Linie 72 - 76**:** Henter og tester iteratoren til den første samling, som det blev gjort for den anden samling i linierne 61 - 66.

Linie 78**:** Tager en kopi af en iterator, alle iteratorklasser i CPH collections kan tage en instans af klassen selv i oprettelsen af nye instanser. Der findes i Python funktionalitet[1] til at få uafhængige iteratorer fra en givet iterator, men denne funktionalitet tager ikke en kopi og typen af de uafhængige iteratorer er ikke den samme som den givet.

Linie 78 - 109**:** Er implementeringen af selve algoritmen.

Linie 111 - 113**:** Afvikler testkoden skrevet i eksemplerne i linierne 41 - 46 og 51 - 59

Antallet a kodelinier, der er unikke for denne algoritme er altså 31 ud af 113 linier, og antallet a linier brugt på at have dokumentationen i kilderkoden er 54.

Listing 6.1: Algoritmen `search` kode; fra `cph_collections/algorithm/search.py`

```
1 """
2 Search for the 1st occuremce of one collection in another.
3
4 """
5 import cph_collections
6
7
8 def search(collection,
```

[1] Itertools.tee : http://docs.python.org/library/itertools.html#itertools.tee

```
9           other,
10          comparator=cph_collections.comparator.equal):
11      """
12      Search for the first appearance of **other** in **collection** using
13      **comparator** to determine whether two elements are the same. An
14      iterator is returned to the first apearance.
15
16      Parameters
17      ==========
18      collection : :ref:`collection`
19        The :ref:`collection` to look in.
20      other : :ref:`collection`
21        The :ref:`collection` to look after.
22      comparator : :ref:`comparator`, optional
23        The :ref:`comparator` to test with. If not specified it defaults to
24        :py:func:`~cph_collections.comparator.equal`
25
26      Returns
27      =======
28      Iterator
29        Iterator pointing to the sequence in **collection** or to the end.
30
31      Raises
32      ======
33      AttributeError
34        If **collection** or **other** is not a known :ref:`collection`.
35
36      Examples
37      ========
38      Create two list objects with elements one list with elements in the
39      other. ::
40
41        >>> import cph_collections
42        >>> a = [1, 2, 3, 4, 5, 6]
43        >>> b = [3, 4]
44        >>> it = cph_collections.algorithm.search(a, b)
45        >>> next(it)
46        4
47
48      Create two list objects with elements one list with elements in the
49      other in reversed order. ::
50
51        >>> import cph_collections
52        >>> a = [1, 2, 3, 4, 5, 6]
53        >>> b = [4, 3]
54        >>> it_b = cph_collections.iterator.backward_iterator(b)
55        >>> it = cph_collections.algorithm.search(a, it_b)
56        >>> next(it)
57        4
58        >>> next(it_b)
59        3
60
61      """
62      it_other = cph_collections.utility.get_iterator(other)
63      if it_other is False:
64          raise cph_collections.error.argument_must_be_a(other,
65                                                         2,
66                                                         'collection')
67
68      # test if collection has a method called search and call it.
69      if hasattr(collection, 'search'):
70          return collection.search(it_other, comparator)
71
72      it_collection = cph_collections.utility.get_iterator(collection)
73      if it_collection is False:
74          raise cph_collections.error.argument_must_be_a(collection,
75                                                         1,
76                                                         'collection')
```

```
77
78      it_other_peeker = it_other.__class__(it_other)
79      try:
80          next(it_other_peeker)
81      except:
82          for e in it_collection:
83              pass
84
85      for element in it_collection:
86          it_other_peeker = it_other.__class__(it_other)
87          it_collection_peeker = it_collection.__class__(it_collection)
88          unbroken = True
89          for e in it_other_peeker:
90
91              if comparator(e, element):
92                  try:
93                      element = next(it_collection_peeker)
94                  except StopIteration:
95                      unbroken = False
96                      try:
97                          next(it_other_peeker)
98                      except StopIteration:
99                          unbroken = True
100             else:
101                 unbroken = False
102                 break
103         if unbroken:
104             return it_collection
105
106     for e in it_collection:
107         pass
108
109     return it_collection
110
111 if __name__ == "__main__":
112     import doctest
113     doctest.testmod()
```

Kapitel 7

C-bindinger

Dette kapitel beskriver C-bindinger. C-bindinger er et forsøg på at implementere dele af CPH collections i C. Ideen med dette er, at teste om C implementeringer er hurtigere end Python implementeringer. Teknikken, der blev brugt, var at implementere elementer ved brug af Python/C API. Grunden til dette valg var, at Python/C API bruges i Python-distributionen. Det antogs dermed, at der ville findes mere hjælp til dette. Følgende kilder blev brugt som inspiration og løsning af problemer:

- Python/C API Reference Manual http://docs.python.org/c-api/index.html. En reference manual.
- Extending and Embedding the Python Interpreter http://docs.python.org/extending/index.html. En guide til, hvordan udvidelser skrives.
- Koden til Python-distributionen http://svn.python.org. Denne kode og i særdeleshed koden til *list*-objektet blev brugt som inspiration til, hvordan C-kode skrives i Python/C API.
- PEP 234 – Iterators http://www.python.org/dev/peps/pep-0234/. En beskrivelse af, hvordan Python-iteratorer bruges i C, dette er nødvendigt for at kunne implementere bibliotekets algoritmer i C.

Følgende erfaringer blev opnået med at skrive implementeringer med Python/C API:

- Det tager væsentligt længere tid at implementere et kodeelement i C og gøre det tilgængeligt for biblioteket i forhold til at implementere det i Python.
- Fejl i en C implementering, der ikke fanges af oversætteren er svære at finde og rette. Python-fortolkeren kan fejle uden fejlbeskeder på fejl i C.
- Der læres ikke C ved at implementere kodeelementer med Python/C API. Største delen af C-koden består i at konvertere til og fra Python værdier og verificere værdierne, der fås fra Python, er de forventede.

	Antal elementer		
Implementering	50100	500100	950100
Python	$0,0729s$	$0,7281s$	$1.3818s$
C	$0.0733s$ $(100,59\%)$	$0.7296s$ $(100,21\%)$	$1.3874s$ $(100,41\%)$

Tabel 7.1: Tiden `count` algoritmen tager i `C` og Python.

- Der læres ikke, hvordan klasser skrives i C++. For at implementere en klasse med `Python/C API`, skrives klassens metoder som frie funktioner, der bliver samlet sammen og gives som en del af en `struct`.

Følgende erfaringer blev opnået med implementeringer skrevet i `Python/C API`:

- Hastighedsforbedringen, der opnås ved at implementere elementer i `C`, er afhængig af, hvor ren implementering er for grænsefladen til Python. For eksempel blev `count` algoritmen forsøgt implementeret i `C`, og som det ses i listing 7.1, er kun linie 8 fri for Python grænsefladen. For at se om `C` implementeringen af algoritmen var hurtigere end Python implementeringen. blev der foretaget en ydelsestest af implemeteringerne. Koden og resultatet af denne test kan ses i bilag F.5 (s. 177) og et uddrag af testresultatet kan ses i tabel 7.1, hvor det kan ses, at `C` implementeringen faktisk er $\frac{1}{2}\%$ langsommere end Python.

  Listing 7.1: `count` implementeret i `C`; fra `cph_collections/include/algorithm_c.c`.

```
1  PyObject* count(PyObject *dummy, PyObject *args){
2      PyObject* iterator;
3      PyObject* arg_element;
4      if (!PyArg_ParseTuple(args, "OO", &iterator, &arg_element)) {
5          return NULL;
6      }
7
8      long c = 0;
9      PyObject* it_element = PyIter_Next(iterator);
10     while(it_element != NULL){
11         c +=  (1 == PyObject_RichCompareBool(arg_element , it_element,
               Py_EQ))?1:0;
12         it_element = PyIter_Next(iterator);
13     }
14
15     /* StopIteration is *implied* by a NULL return from
16      * PyIter_Next() if PyErr_Occurred() is false.
17      */
18     if (PyErr_Occurred()) {
19         if (PyErr_ExceptionMatches(PyExc_StopIteration)){
20             PyErr_Clear();
21         }
22     }
23     return PyInt_FromLong(c);
24 }
```

- `C` implementeringer kan være hurtigere end deres tilsvarende Python implementering, for eksempel er en stak implementeret over en enkeltlinket

	Antal elementer		
Implementering	50100	500100	950100
Python	$0,0931s$ (3458%)	$0.7296s$ (3533%)	$1,7476s$ (3420%)
C	$0,0027s$	$0,0261s$	$0,0511s$

Tabel 7.2: Tiden det tager at gennemløbe en stak implementeret i C og Python.

ring i C hurtigere end den tilsvarende implementering i Python. I CPH collections findes begge implementeringer, og der blev foretaget en test af, hvor hurtigt elementerne i de to implementeringer kunne gennemløbes og summeres. Koden og resultatet af testen kan ses i bilag F.6 (s. 179) og et uddrag af resultatet kan ses i tabel 7.2 (s. 104).

Mens det kan ses, at stakken i C er hurtigst, så er det ikke nok kun at implementerer stakbeholderen; eftersom stakbeholderen ikke kan itereres gennem dens grænseflade, og da objekt variabler i C implementeringer ikke kan fås i Python, er det nødvendigt også at implementere iteratorerne til stakken i C. Så om det reelt er C stakken eller C iteratoren, der er skyld i besparelsen. er uvist.

- Elementer i C skal kompileres. Dette rejser spørgsmålet om, der skal laves en kompileret version af C-delen af biblioteket for alle tænkelige platforme. Eller om det skal kræves, at brugerne selv kompilerer elementerne og i tilfælde af det sidste, om biblioteket skal have Python implementeringer, der kan bruges i stedet for C implementeringer, når disse ikke er kompilerede.

- En bedre og simplere måde at forbedre køretiden for biblioteket kunne være brug af værktøjer, der forbedrer køretiden af Python-kode. Et sådan værktøj kunne være psyco[1].

Der findes alternativer[2] til at implementere løsninger i C-moduler end Python/C API. Det er formentlig lettere at bruge disse, da de kan hjælpe med at abstrahere væk fra selve grænsefladen mellem C og Python til at skrive C-kode eller C++-kode. Et andet interesant alternativ er *Weave*[3] med *Weave* er det muligt at skrive C-kode direkte i Python-koden som en tekststreng.

Effekten af at kunne skrive C-kode i Python er, at det er muligt kun at skrive den del af en algoritme, der skal optimeres i C – for eksempel løkker – og skrive resten i Python – for eksempel test af om inddata er korrekt –, men det betyder dog, at C-koden i formentlig alle koderedigeringsværktøjer vil være uden syntaksfarvning og bare ligne en tekststreng.

Grunden til, at ingen af disse alternativer blev brugt eller forsøgt brugt er, at de alle udover en oversætter også kræver et tredjepartsværktøj. Dette krav ville betyde, at udviklere af CPH collections og formentlig også brugerne skulle hente endnu et værktøj for at bruge CPH collections.

[1] Psyco: http://psyco.sourceforge.net/.html
[2] Liste af alternativer: http://wiki.cython.org/WrappingCorCpp
[3] Weave:http://www.scipy.org/Weave

Kapitel 8

Testning

Dette kapitel beskriver testningen, der foretages af komponenterne i CPH collections. En del af testningen af komponenterne består i eksemplerne skrevet i dokumentationen, der alle er blevet testet at de virker. Eksemplerne er blevet testet med værktøjet *Doctest*, som beskrevet i sektion 2.1.11 (s. 12), men udover eksemplerne er der skrevet modultest (*unittest*) af alle de komponenter (algoritmer og beholderer), der er tilgængelige for brugerne.

Det er altså de størst tænkelige komponenter af CPH collections der testes og ikke de interne komponenterne, der indgår i opbyggelsen af dem. Grunden til dette er fra [7], der forslår kun at skrive modultest til de større moduler.

Modultestene, der foretages i CPH collections, er sort bokstests [4], hvilket betyder, at modultesten kun tester grænsefladen, der er tilgængelig for brugeren og ikke garanterer, at hele modulets kode er testet. Hver test består i, at teste om et komponent givet inddata resulterer i det forventede uddata, eller i tilfælde af objekter, efterlader objektet i den forventede tilstand.

Det er let at finde inddata, der ikke resulterer i en fejl, men at finde den ene kombination af inddata (grænsetilfældet), som resulterer i fejl er svært. For eksempel er grænsetilfældet ved vektorens `push_back`-metode ikke kun, når der er 1 eller 0 elementer i vektoren, men også når `push_back`-metoden aktiverer vektoren formindskningspolitik. Ligeledes skal det overvejes, hvad objektets forventede tilstand er efter et metodekald. For eksempel forventes det efter et kald til `pop_back`-metoden ikke kun, at antallet af elementer falder med 1, hvis vektoren ikke er tom, men også at det er muligt efterfølgende at fjerne og lægge elementer i vektoren.

Modultest skrives, så de kan afvikles med værktøjet *unittest*[1], der følger med Python. Værktøjet kan sættes til at skanne[2] en folder efter Python-filer, hvis navne starter med "test". Disse filer fortolkes, og der oprettes instanser af klasser, hvis navne starter med "test" i filerne. Hvis klasserne arver fra `TestCase` klassen vil denne oprettelse indbefatte kald til alle metoder, der starter med "test" i klassen. (Koden i metoderne er modultestene og klassen er en samling

[1] unittest: `http://docs.python.org/library/unittest.html`
[2] Kommando: python -m unittest -discover

af modultest.) Dette værktøj gør det dermed let at teste ofte og teste alt, som foreslået i [7]. Grunden til at teste alt og teste ofte er, at desto større kodebasen er, desto mere uoverskuelig bliver konsekvenserne af kodeændringer et sted, og derfor testes hele kodebasen i stedet for kun de dele af kodebasen der menes at være blevet ændret. dette kaldes regressionstestning.

I implementeringen af modultestene bruges rådende fra [7] om at bruge *MockObject*s og om at skrive hjælpe funktionaliteter, der kan bruges på tværs af de forskellige modultest.

Det første råd er brugt til at lave et element objekt, som er det der bliver lagt i instanser af beholderne. Grunden til, at der ikke bare blev testede med et simpelt element som et tal, er at dette tal ikke ville kunne teste om en beholder eller algoritme gjorde brug af kopieringer, som de i de fleste tilfælde skal, eller om referencerne de gives bruges. For eksempel skal algoritmen *remove* rykke alle elementer, der er efter det fjernede frem. Dette kan gøres ved at rykke deres referencer, men for det sidste element skal der tages en kopi, da der elles vil opstå to referencer til det samme objekt. At algoritmen udføre dette kan testes ved at ændre i det sidste *MockObject* i beholderen efter algoritmen er færdig, denne ændring skal ikke have nogen effekt på det næst sidste objekt.

Det andet råd er brugt til at lave standard funktioner til at fylde en beholder med elementer og eventuelt specifikke elementer, dette i stedet for at skrive koden til at fylde beholderne i hver modultest. Rådet er brugt i testen af beholdernes grænseflader dette ved, at modultestene af en beholders grænseflade er skrevet i en klasse og beholdernes modultest skal da blot arve fra denne fælles klasse.

Kapitel 9

Ideer til fremtiden

Dette kapitel beskriver ideer til fremtidige udviklinger eller eksperimenter, der kan foretages for CPH collections. Kapitlet beskriver også fejl og mangler i den nuværende udgave af biblioteket.

9.1 Blokallokering i `C`

At kunne allokerer en blok af hukommelse til objekter i `C`. Dette vil gøre det muligt bedre at kunne eksperimentere med implementeringer af forskellige datastrukturer baseret på blokke af elementer.

I CPH STL[1] er der for eksempel forskellige realisationer til vektoren, der alle er baseret på udvidelses- og formindskningspolitiker. Disse politiker allokerer eller frigiver hukommelseblokke af forskellige størrelser.

Uden mulighed for at allokere en specificeret størrelse af blokke må eksperimenter bruge en datastruktur som Python-list til at emulere en blok, men denne listes størrelse kan ikke fastsættes.

9.2 Ensartet iteratoropførsel

I CPH collections bruges både iteratorer baseret på indeks iterering af en beholders grænseflade, og iteratorer baseret på iterering af en beholderes interne iterable data. Mens dette ikke skaber nogen forskel i det fleste tilfælde, så er der en forskal når elementerne i to beholdere ombyttes. I dette tilfælde vil den første iteratortype forsatte iterering være over de nye elementer i den samme beholder, men den anden iteratortype forsatte iterering vil være over de samme elementer, men i den nye beholder.

For at gøre dette ensartet, kan beholdere og iteratorer af den første type bruge observeringsmønstret (*observer pattern*) fra [6]. Beholderen overvåger sine iteratorer, og ved ombytning af elementerne beholderen nliver iteratorerne

[1] CPH STL: `http://www.cphstl.dk`

ombyttet og fortalt om denne ombytning, så de kan itererer i beholderen, der nu indeholder elementerne iteratoren itererede over.

Det omvendte kunne også opnås, hvor den anden iteratortype blev observeret, og ved ombytning blev sat til at itererer fra det samme element nummer, som iteratoren var ved i de foregående elementer. Denne løsning ville dog, afhængig af iteratoren, kræve en iterering fra det første element indtil det nye element.

9.3 Universel iteratorgrænseflade

Python iteratorgrænsefladen, se figur 5.1 (s. 67), er ikke optimal til implementeringen af C++0x specifikationen [2]. I den nuværende udgave af CPH collections er nogle af iteratorerne udvidet med flere metoder i deres grænseflader.

- Det er muligt at få det nuværende element en iterator peger på i de associative beholdere.

  Grunden til dette er, at de associative beholdere har metoder som `lower_bound`, der returnerer en iterator. Den returnerede iterator peger på et element med den givet nøgle eller større end den givet nøgle og for, at brugerne eller algoritmer skal kunne finde ud af hvilket element iteratoren peger på er det muligt at få dette gennem iteratorens grænseflade.

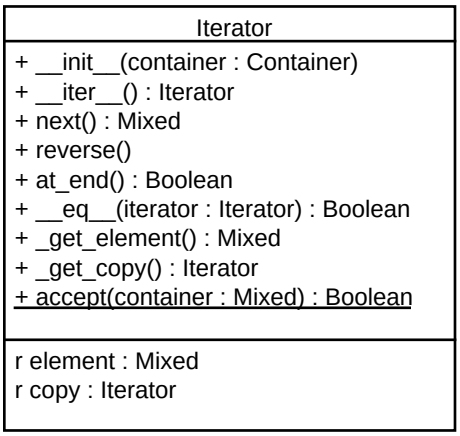
- For iteratorerne til associative beholdere er det muligt at finde ud af om iteratoren er ved enden af itereringen. Dette kan opnås ved forsøg på, at få elementet iteratoren peger på, hvis den er ved enden vil forsøget rejse en undtagelse. Dette er ikke muligt for andre iteratorer.

- I visse tilfælde er det nødvendigt at kunne finde ud af om to iteratorer peger på det samme element. For eksempel i algoritmen `rotate`, men eftersom dette ikke er muligt, så må algoritmen selv tælle hvor mange elementer, den har rykket en iterator.

- Da iteratorer kun har `next` metoden til at rykke på, er de nødvendigvis envejs iteratorer. Det kunne dog tænkes, at det i visse tilfælde ville være brugbart at kunne vende iteratoren om. Dette kan opnås for visse iteratortyper ved at give iteratoren til konstruktøren af den omvendte iteratortype.

- I visse tilfælde er det nødvendigt at iterere fremad, men stadig kunne returnere til det oprindelige sted. Et eksempel på dette er i algoritmen *search*, hvor der undersøges om en sekvens er i en anden sekvens af elementer. I disse tilfælde er det nødvendigt at kunne tage en kopi af iteratoren.

I stedet for at have visse iteratorer, der kan noget, som andre ikke kan, ville det være en ide at implementere iteratorerne så de alle kunne det samme med den samme universelle grænseflade. Figur 9.1 (s. 109) viser klassediagrammet for et forslag til en sådan grænseflade.



Figur 9.1: Udvidet iteratorgrænseflade.

Følgende liste beskriver elementerne i den foreslået udvidelse af iterator grænsefladen.

`__eq__` : Givet en iterator af samme type bruges denne metode til at teste om de to iteratorer peger på det samme element i den samme beholder.

copy og `_get_copy`: Returnerer en kopi af iteratoren; `copy` er en Python-egenskab, der kalder metoden `_get_copy`.

element og `_get_element`: Returnerer elementet iteratoren er ved; `element` er en Python-egenskab, der kalder metoden `_get_element`, hvis iteratoren ikke er ved et element, rejses en undtagelse.

reverse : Metoden `reverse` vender iteratoren til at iterere den modsatte vej eller rejser en undtagelse, hvis iteratoren ikke kan vendes.

at_end : Returnerer sandt hvis iteratoren er ved enden af den iterering.

Et sidste forslag til at udvide grænsefladen er at lade det være muligt at få iteratoren til ikke at rejse undtagelsen `StopIteration` og få den til det igen. Denne del af Pythons iteratorkoncept er pænt så længe iteratoren bruges i en for-løkke, der selv fanger undtagelsen, men det bliver et virvar af `try except` blokke, når iteratoren bruges uden for en for-løkke.

9.4 Frie iteratorer

Den nuværende udgave af CPH collections tillader en hvilken som helst type af iterator, men, hvis der aldrig bruges andet end forlæns- og baglænsiteratorer, ville det formentlig være mere brugervenligt og simplere ikke at have denne frihed.

Hvis de eneste iteratorer der bruges, er forlæns- og baglænsiteratorer, så er bindingen af iteratorer til grænsefladen i C++0x specifikationen [2] bedre, da denne er mindre kompleks end det nuværende frie iterator koncept brugt i CPH collections.

9.5 Review af dokumentation

Ligesom med kode så bør dokumentation gennemlæses og rettes i reviews for at forbedre kvaliteten [14]. Dokumentationen skrevet i CPH collections har ikke været udsat for nogen organisaseret gennemgang eller brugerundersøgelse.

En brugerundersøgelse kunne være brugbar for at finde ud af, om dokumentationen overhovedet hjælper brugerne, og et review eller brugerundersøgelse kunne også finde stave- og grammatikfejl.

9.6 Intelligente fabrikker

De nuværende udgaver af beholderfabrikkerne har ikke nogen intelligens; de returnerer simpelthen den udgave af en beholder, som brugeren beder om. Dette kræver, at brugerne kender til de enkelte implementeringer af beholderen, og for at foretage et intelligent valg, kender til deres fordele og ulemper.

For at fjerne dette krav fra brugerne kunne fabrikkerne udbygges med viden om fordelene og ulemperne ved de enkelte beholderimplementeringer. Dette kunne for eksempel implementeres som resultatet af benchmarktest på implementeringerne.

Hvis fabrikken ved hvad dens beholdere er gode til kunne brugerne nøjes med at fortælle fabrikken, hvad en implementering skal være god til, eller endnu bedre fabrikkerne kunne registrere den faktiske brug af beholderen de giver og returnere den bedste beholder. (Ved at lægge en dekorator, der registrerer beholderens brug, om den returnerede beholderinstans kunne fabrikken finde ud af, hvordan en beholder bliver brugt.)

9.7 Universelle beholdere

I stedet for at brugeren bevidst skal vælge mellem en specifik sekvensbeholder (vektor, liste, etc..) eller en specifik assosiativbeholder (sæt, kort, etc..). Kunne det tænkes, at der kunne implementeres en universel sekvens- eller assosiativbeholder (broer), der ud fra brugen af beholderen valgte den bedste realisation for brugeren. (Da det er en bro, kan alle kald til den registreres.)

For eksempel, hvis direkte indeksering kun bruges en gang imellem, men der ofte indsættes elementer i midten skulle broen vælge en liste, og hvis direkte indeksering bruges ofte skulle den vælge en vektor. Bliver der kun indsat og fjernet elementer i den ene ende kunne den bruge en enkeltlinket liste eller evt. en omvendt vektor.

9.8 Balancerede træer

I CPH collections findes kun et standard binært søgetræ, der ikke indholder nogen politiker om træets balance. En mulig udvidelse af biblioteket ville derfor være med træer, der havde politiker på dette område.

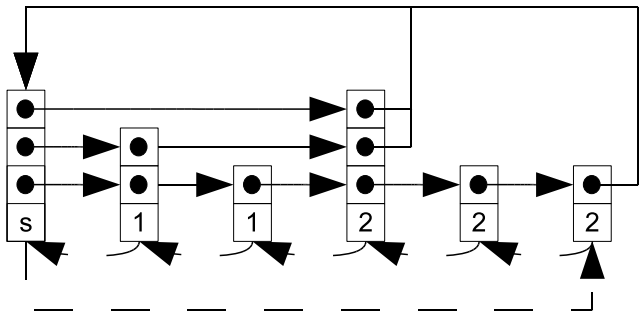
Eventuelt eftersom dette kun har effekt på indsættelse og fjernelse af elementer fra træet, kunne disse politiker implementeres som strategier, og der kunne kodes en implementering af træet, der accepterer disse strategier.

9.9 Multisæt og -kort implementeringer

De nuværende implementeringer af multisæt og -kort opnår, at antallet af sammenligninger de foretager for at finde en nøgle, er det samme, som hvis de var et sæt eller kort, men dette opnås ved kompleksiteten i at have to datastrukturer; en liste for at holde alle ens elementer og en sorteret beholder til disse lister. Effekten af denne implementering er, at det kræver en mere kompleks iterator, der kombinerer iteratorene af de to underlæggende datastrukturer.

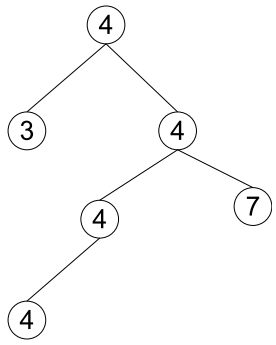
En simplere implementering af disse beholdere ville være, at den underlæggende sorterede beholder havde politiker, der placerede ens elementer, så de ikke krævede flere sammenligninger.

En sådan politik for skiplisten kunne være, at alle ens elementer havde niveau 1 undtagen det første af disse elementer, der mindst skulle have niveau to. Søgninger efter en nøgle i listen kunne da ignore niveau 1, se figur 9.2.



Figur 9.2: En skipliste, hvor søgning efter en nøgle højst tester ens elementer én gang.

En sådan politik for et binært søgetræ kunne være, at ved to ens elementer bliver det andet element indsat som højre barn til det første, og alle efterfølgende ens elementer er venstre børn af det andet element. Effekten af denne opbygning er, at en søgning efter en nøgle højest tester ens elementer to gange, se figur 9.3 (s. 112).



Figur 9.3: Et binært søgetræ, hvor søgning efter en nøgle højst tester ens elementer to gange.

9.10 Dekoratorer

Der mangler base- og kopieringdekoratorer til alle beholdertyper undtagen vektor og sæt. Endvidere kunne der implementeres uforanderlige dekoratorer til de associative beholdere. Disse er krævet for at opfylde uforanderligheden i de beskrevne koncepter, denne uforanderlighed er ikke implementeret i beholderne under antagelsen, at beholderne bruges korrekt, sparer denne manglende implementering resourcer.

9.11 Iterator samlingsbeholder

Der mangler en samling, der reelt set defineres, som alle elementer mellem to iteratorer. Den første iterator skal kunne itereres mod den anden, og den anden iterator skal eventuelt kunne itereres mod den første. Samlingen skal kunne oprettes givet to iteratorer, men eventuelt også givet en iterator og et tal, hvor tallet er antal af skridt fra den givne iterator til den ikke givne iterator.

Instanser af denne samlingstype skal kunne gives til algoritmer og metoder, som enhver anden samling. Samtidig skal iteratorer ikke længere opfattes og accepteres som samlinger.

Dette ville kræve mindre ændringer i den nuværende kodebase, og at brugere, der før gav en iterator som en samling først opretter en instans af denne beholdertype og giver denne til algoritmen. Det er endvidere nødvendigt som minimum at udvide iterator grænsefladen, så to iteratorer kan sammenlignes.

9.12 Algoritmer

I den nuværende udgave af CPH collections findes mange af algoritmerne fra C++0x specifikationen [2], men ikke dem alle. Tabel 9.1 (s. 113) lister navnene

all_of	any_of	copy_backward
copy_if	copy_n	equal_range
fill_n	find_end	find_first_of
find_if_not	for_each	generate
generate_n	inplace_merge	is_heap
is_heap_until	is_permutation	is_partition
is_sorted	is_sorted_until	iter_swap
max	max_element	min
min_element	minmax	move
move_backward	next_permutation	none_of
nth_element	partial_sort	partial_sort_copy
partition_copy	partition_point	prev_permutation
replace	replace_copy	replace_copy_if
replace_if	remove_copy	remove_if
swap_ranges	transform	unique_copy

Tabel 9.1: Alfabetisk listning af manglende algoritmer i CPH collections.

på de manglende algoritmer. En indlysende udvikling af CPH collections ville være at implementere disse algoritmer.

9.13 Egenskabsprogrammering

I beskrivelsen af koncepterne til de forskellige beholdertyper blev der beskrevet, hvilke egenskaber beholderne har. Men i grænsefladen til beholderne blev egenskaberne implementerede som metoder og Python-egenskaber.

Et muligt eksperiment vil være at udvide Python så mere end `get`, `set` og `del` kan sættes i en Python-egenskab. For eksempel kan vektorkonceptets ende hentes, sættes, fjernes og lægges til. Hvis Python blev udvidet så det var muligt at skrive erklæringer med egne nøgleord kunne det gøres muligt at kode som i listing 9.1, og hvis det kunne gøres muligt at give flere værdier til en egenskab kunne vektorens `resize` metode kobles til `size` egenskaben.

Listing 9.1: Egenskabsprogrammering

```
1  v = Vector()
2  v.back = 5
3  e = v.back
4  pop v.back
5  v.back push 5
6  v.size = 5 default element 3
7
8  #Men den nuværenge implementering af egenskaber kan kun følgende bruges.
9  v.back = 5
10 e = v.back
11 del v.back # Denne er ikke i CPH collections.
```

Alternativt kunne andre operatorer (`*`, `/`, `@`) registreres til at kalde en metode når de blev kaldt med eller på egenskaben, men dette ville fjerne operatorernes funktionalitet på den værdi, som egenskaben returnerede.

Et problem med at udvide Python-egenskaber til at være mere fleksible er, at det udvider den viden, som en bruger skal have om egenskaben. Det er et reelt spørgsmål, om det er bedre for brugeren at have en egenskab, der med sikkerhed højest forstår tre ting og lade resten være i metoder eller at have en egenskab, der kan alt.

Litteratur

[1] S. Anderson, The factory pattern in Python with __new__, WebDokument (2009). Tilgængelig på http://whilefalse.net/2009/10/21/factory-pattern-python-__new__/.

[2] ISO, Programming languages - C++, Webside tilgængelig på http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf (2010).

[3] J. Bloch, How to design a good API and why it matters, WebDokument (2005). Tilgængelig på http://lcsd05.cs.tamu.edu/slides/keynote.pdf.

[4] M. Clifton, Advanced unit testing, part i - overview, WebDokument (2003). Tilgængelig på http://www.codeproject.com/KB/cs/autp1.aspx.

[5] M. Ettrich, Designing Qt-style c++ APIs, WebDokument (2005). Tilgængelig på http://lcsd05.cs.tamu.edu/slides/keynote.pdf.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).

[7] A. Iskold, 12 unit testing tips for software engineers, WebDokument (2008). Tilgængelig på http://www.readwriteweb.com/archives/12_unit_testing_tips_for_software_engineers.php.

[8] Python, Iterators, Webside tilgængelig på http://www.python.org/dev/peps/pep-0234/ (2007).

[9] A. Langer, Invalidation of stl iterators, WebDokument (2002). Tilgængelig på http://www.angelikalanger.com/Conferences/Slides/CppInvalidIterators-DevConnections-2002.pdf.

[10] Python, Simple generators, Webside tilgængelig på http://www.python.org/dev/peps/pep-0255/ (2009).

[11] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, *Commun. ACM* **33**, 6 (1990), 668–676.

[12] Python, Pythonspeed / performancetips, Webside tilgængelig på http://wiki.python.org/moin/PythonSpeed/PerformanceTips (2010).

[13] Python, Style guide for Python code, Webside tilgængelig på `http://www.python.org/dev/peps/pep-0008/` (2009).

[14] I. Sommerville, Software documentation, WebDokument (2001). Tilgængelig på `http://www.literateprogramming.com/documentation.pdf`.

[15] J. Tulach, *Practical API Design*, Apress (2008).

[16] Object Management Group, UML v1.1, notation guide, Webside tilgængelig på `http://www.cs.wustl.edu/~kjg/cse132/forms/UMLnotationguide.pdf` (1997).

[17] Wikipedia, Class diagram, Webside tilgængelig på `http://en.wikipedia.org/wiki/Class_diagram` (2010).

Bilag A

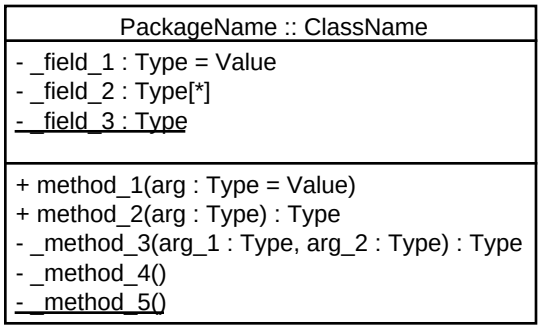
UML-notationen

Dette bilag beskriver UML-notationen brugt i specialet. Notationen følger specifikationen fra guiden [16], der ikke er den nyeste version af UML.

A.1 Klasser

I UML illustreres en konkret klasse med elementet vist i figur A.1. Elementet er delt op i 4 bokse, indeholdet af de enkelte bokse er som følger.

- I den øverste boks står navnet (`ClassName`) på den pågældende klasse, hvis nødvendigt kan der foran navnet skrives, hvilken pakke (`PackageName`) klassen er fra. Navnet på pakken og navnet på klassen adskilles af ::.
- I den anden boks listes klassens variabler (*fields*). En variabel listes med dens tilgængelighed (*scope*), navn, type, antal og forudbestemte (*default*) værdi.



Figur A.1: Elementet for en konkret klasse.

Tilgængeligheden af en variabel illustreres med tegnene `+` og `-` for henholdsvis offentlig (*public*) og privat .

Variablens navn og type adskilles med `:`, hvis variablen dækker over flere instanser af den pågældende type illustres dette med `[n]` efter typen, hvis det er et fast antal instanser skrives antallet i stedet for `n` ellers skrives `*` i stedet for `n`.

Et eksempel på en variabel, der dækker over flere elementer, er array af elementer. I dette tilfælde vil variablens type blive skrevet som elementets type efterfulgt af `[*]`, dette er teknisk set forkert, da typen er array.

Hvis variablen har en forudbestemt værdi, illustreres dette med et `=` efterfulgt af værdien.

Hvis en variabel er en klassevariabel, hvilket vil sige, at den er tilgængelig gennem klassen og ikke gennem instanser af klassen, illustreres dette ved, at der tegnes en bundlinie under hele variablens illustration.

- I den tredje boks listes klassens metoder, hver metode listes med dens tilgængelighed, navn, argumenter og typen af den værdi der returneres. En metodes tilgængelighed illustreres på samme måde som for variabler.

  Argumenterne til en metode omkranses af et parentespar og adskilles med kommaer. (En metode, der ikke tager nogen argumenter, har et tomt parentespar.) Hvert argument skrives med dets navn[1], type og forudbestemte værdi, disse illustreres på samme måde, som de blev for variabler.

  Hvis metoden returnerer en værdi illustreres dette med `:` efterfulgt af værdiens type. (Alle metoder returnerer en værdi, men returtypen udelades for metoder, der ikke specifikt har en `return`-erklæring.)

- I den fjerde boks listes klassens Python-egenskaber, hver egenskab listes med dens tilgængelighed, navn og type. (Denne boks er ikke en del af `UML` specifikationen.) Tilgængligheden af en egenskab er dels, om den er offentlig eller privat, men også om den kan læses (*read*), skrives (*write*) og slettes (*delete*).

  Den offentlige og private tilgængelighed illustreres som ved variabler, og hvis egenskaben kan læses, illustreres dette med et `r`, hvis egenskaben kan skrives illustreres dette med et `w`, og hvis den kan slettes illustreres dette med et `d`. En egenskabs navn og type illustreres som ved variabler.

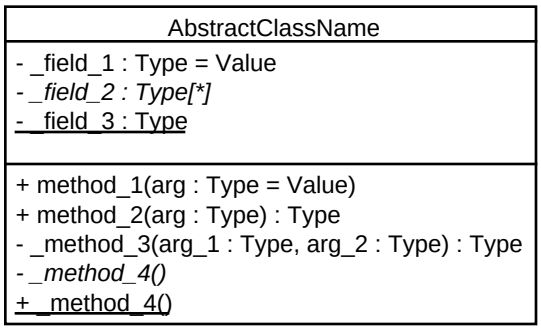
  Hvis en metode er en klassemetode, hvilket vil sige den er tilgængelig gennem klassen og ikke gennem instanser af klassen, illustreres dette ved, at der tegnes en bundlinie under hele metodens illustration.

[1] Dette er relevant da Python tillader, at værdier bindes til argumentets navn i metodekaldet.

A.2 Abstrakt klasse

Elementet i figur A.2 illustrerer en abstrakt klasse. Abstrakte klasser er illustrativt ens med konkrete klasser, den eneste forskel er, at klassens navn og de abstrakte elementer i klassen skrives med kursiv skrift.

Hvis en klasse A arver fra en abstrakt klasse B, men ikke implementerer de abstrakte elementer i B er klassen A også abstrakt, og derfor kan der findes abstrakte klasser uden abstrakte elementer i et `UML`-diagram.



Figur A.2: Elementet for en abstrakt klasse.

A.3 Simpel klasse

Hvis en eller flere af de tre sidste bokse i et klasseelement ikke er nødvendige i et diagram, kan de udelades fra elementet. Dette gælder for både konkrete- og abstrakte klasser.

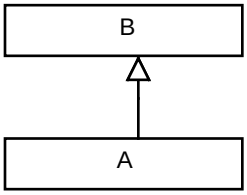
Eftersom elementerne i de tre bokse ikke illustreres ens er det muligt at genkende de enkelte boks, hvis andre udelades. Figur A.3 viser, hvordan klasseelementerne ser ud når, boksene for variabler, metoder og egenskaber udelades.

Figur A.3: Konkret og abstrakt klasseelementer uden variabler, metoder og egenskaber.

A.4 Arvning mellem klasser

Arvning mellem to klasser illustreres med en pil, hvis spids er tom. Spidsen peger på klassen, der arves fra, og enden på klassen, der arver.

Placeringen af klasseelementerne er ikke relevant, det er udlukkende pilen, der indikerer arv, men den almindelige placering er at placere klassen, der arves fra over klassen, der arver. Figur A.4 viser en konkret klasse `A`, der arver fra en abstrakt klasse `B`.



Figur A.4: Arvning fra en abstrakt klasse til en konkret klasse.

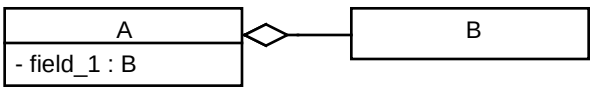
A.5 Aggregering mellem klasser

Aggregering (*Aggregation*) mellem to klasser illustreres med en linie, der ender i en tom diamant. Figur A.5 viser en aggregering mellem klasserne `A` og `B`. Den tomme diamant peger på klassen `B`, der aggregerer, og den anden ende peger på klassen `B`, der aggregeres.

Aggregering betyder at en instans af klassen `A` har tilgang til en eller flere instanser af klassen `B`, så længe instansen eksisterer, men også samtidig at b ikke ophører med at eksistere når a gør det.

Hvis der er tale om et ukendt antal af instanser `B` illustreres dette med `*` på den ende af linien der peger på `B`. Hvis der ikke står noget på linien, er der tale om èn instans af `B`.

Beholdere, der holder på noder, er illustreret med aggregeringer, eftersom noderne ikke ophører med at eksistere når beholderen forsvinder.

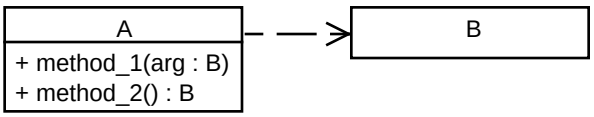


Figur A.5: Aggregering mellem to klasser `A` og `B`.

A.6 Afhængighed mellem to klasser

Afhængighed *(*Dependency) mellem to klasser `A` og `B` illustreres med en stiplet pil der peger fra den klasse, der er afhængig af eksitensen af en anden klasse til denne klasse. På figur A.6 er klassen `A` afhængig af klassen `B`.

Afhængighed opstår, når en klasse i sine metoder gør brug af instanser af en anden klasse eller klassen selv. Denne afhængighed i en metode opstår, hvis argumenterne til metoden (`metode_1`), lokale variabler i metoden, eller returværdien fra metoden (`metode_2`) er en instans af en anden klasse [17].



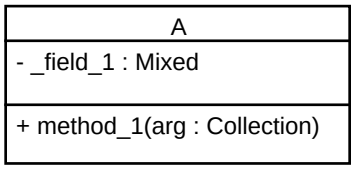
Figur A.6: Afhængighed mellem to klasser `A` og `B`.

A.7 Pseudotyper

Figur A.7 viser pseudotyperne `Mixed` og `Collection`, disse typer findes ikke i CPH collections, men de bruges i `UML`-diagrammerne til at indikere følgende.

Pseudotypen `Mixed` indikerer, at typen kan være en hvilken som helst type, for eksempel er der ikke noget krav til typen af elementer i en beholder, derfor bliver denne type beskrevet som `Mixed`.

Pseudotypen `Collection` indikerer, at der forventes en instans, der er en samling af elementer og som kan itereres.



Figur A.7: Pseudo typer brugt i UML i dette speciale.

Bilag B

Beskrivelse af metoderne i de abstrakte datatyper

I dette bilag bliver metoderne og egenskaberne i de abstrakte datatyper uformelt beskrevet. Beskrivelserne beskriver hvad formålet med den enkelte metode eller egenskab er.

Når der refereres til kompleksiteten af metoderne skal dette kun ses som vejledende, og en implementering af et koncept er ikke tvunget til at opfylde kompleksiteten. Kompleksiteterne er fra C++0x specifikationen [2]. Kompleksiterne kan som regel oversættes til køretid, hvor en konstant kompleksitet betyder at tiden en metode tager ikke er påvirket af antallet af elementer den opererer på, og er kompleksiteten linæer i et antal af n kan det forventes at køretiden stiger med en tilnærmelsesvis konstant faktor når n stiger. Er kompleksiteten højest linæer i et antal, er dette i værste tilfælde.

Selvom mange af koncepterne har metoder og egenskaber, der er ens på tværs af koncepterne, så vil hver metode og egenskab blive beskrevet for hvert koncept. Dette er i stedet for at beskrive dem et sted og lade det være op til læseren at navigere til dette sted og tilbage igen.

Når der i dette bilag er referencer til specifikationen, menes der C++0x specifikationen [2].

B.1 Vektor

Den efterfølgende liste beskriver metoderne og egenskaberne i vektorkonceptet.

`__init__`: Dette er konstruktøren af instanser af vektortypen, den skal kunne tage en samling af elementer, men også kunne acceptere ikke at få en samling. Kopier af elementerne i samlingen lægges i vektoren. Den forventede kompleksitet af denne metode er lineær i antallet af elementer i den givne samling.

Under oprettelsen af vektoren skal den ikke foretage reallokeringer af elementerne i den.

`__del__`: Dette er destruktøren, den skal sørge for at frigive elementerne i vektoren, før vektoren selv frigives. Den forventede kompleksitet er lineær i antallet af elementer i vektoren.

Under nedlæggelse af vektoren skal den ikke foretage reallokeringer af elementerne i den.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen vektoren har er den eneste til elementet vil Python selv frigive elementet. (Garbage collector.)

`capacity`, `_get_capacity` og `_set_capacity`: Handler alle om kapaciteten af vektoren, dette er antallet af elementer vektoren, kan have eller skal kunne have.

Egenskaben `capacity` returnerer eller sætter kapaciteten, denne egenskab kalder `_get_capacity` for at returnere kapaciteten og `_set_capacity` for at sætte kapaciteten til den ønskede kapacitet.

Den forventede kompleksitet for at få kapaciteten er konstant, og den forventede kompleksitet for at sætte kapaciteten er højest lineær i antallet af elementer i vektoren.

Hvis kapaciteten, der sættes, forårsager at vektoren reallokerer elementerne er kompleksiteten lineær i antallet af elementer.

`swap`: Handler om at ombytte elementerne mellem to vektorer af samme implementering.

Den forventede kompleksitet er konstant, da det forventes, at dette kun kræver ombytning af et konstant antal referencer.

`__len__`, `size`, `_get_size` og `resize`: Handler om at få antallet af elementer i vektoren.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af elementer, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af elementer i vektoren er konstant.

Metoden `resize` ændrer antallet af elementer til et givet antal. Metoden gives antal af elementer og værdien nye elementer skal have. Gives ingen værdi er den `None`. Den forventede kompleksitet er højest lineær med endelige antal af elementer. Grunden til en lineær kompleksitet er, at ændringer i antallet af elementer kan forårsage reallokering af alle elementer i vektoren.

Egenskaben `size` kan ikke bruges til at sætte antallet af elementer i vektoren. Grunden til dette er, at en egenskab kun kan tildeles en værdi, men ved sætning af antallet af elementer skal bruges både størrelsen og et element, der skal kopieres. Dette kan godt være blevet implementeret

for eksempel kunne en bruger kombinere de to værdier til et par, som en metode så kunne adskille igen.

`back`, `_set_back`, `_get_back`, `pop_back` og `push_back`: Handler om at ændre det sidste element i vektoren.

Egenskaben `back` kalder metoderne `_get_back` og `_set_back` for henholdsvis at returnere og sætte det sidste element i vektoren. Den forventede kompleksitet for at sætte eller hente et element er konstant.

Metoden `push_back` lægger er element til sidst i vektoren, den bruges også af `_set_back` når vektoren er tom. Metoden `pop_back` fjerner det sidste element i vektoren. Den forventede kompleksitet for begge metoder er højest lineær med antallet af elementer, da de kan forårsage en reallokering af alle elementerne i vektoren. At `_set_back` bruger denne metode på en tom vektor ændrer ikke på, at `_set_back` forventes at have en konstant kompleksitet. (Den kan ikke forårsage en reallokering af 0 elementer.)

`front`, `_set_front`, `_get_front`: Handler om at ændre det første element i vektoren.

Egenskaben `front` kalder metoderne `_get_front` og `_set_front` for henholdsvis at returnere og sætte det første element i vektoren. Metoden `push_back` kaldes af `_set_front`, hvis vektoren er tom. Den forventede kompleksitet af dette er konstant.

`clear`: Frigiver elementerne i vektoren, men frigiver ikke vektoren selv. Den forventede kompleksitet for dette er lineær i antallet af elementer i vektoren.

Under frigivelsen af elementerne må der ikke foretages reallokeringer af elementerne i den.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen vektoren har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`__iter__`: Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første element i vektoren til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`, `_get_is_empty` og `__non_zero__`: Handler om at teste om vektoren er tom eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt, hvis vektoren er tom, og metoden `__non_zero__` returnerer sandt, hvis vektoren ikke er tom, denne metode er en del af Python, og kunne udelades da Python tjekker på, hvad `__len__` returnerer, når metoden ikke findes. Den forventede kompleksitet af metoderne er konstant.

`__getitem__` og `__setitem__`: Disse to metoder tager begge et indeks (tal) som det første argument og den sidste metode tager en værdi som det andet argument. Den første metode returnerer elementet på det givne indeks og den anden sætter elementet på det givne indeks. Disse metoder gør, at vektoren kan blive indekseret som et array, som specifikationen foreskriver. Den forventede kompleksitet af metoderne er konstant.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` og `__eq__`: Handler om at sammenligne to vektorerer med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem elementerne i vektorerne i deres orden.

Metoderne implementerer sammenligningsoperatorerne for vektorinstanser så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to vektorer. Den forventede kompleksitet er højest lineær i antallet af elementer i vektoren med færrest elementer.

`__repr__` og `__str__`: Handler om at få en tekstrepræsentation af vektoren.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af vektorens nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i vektoren, hvis repræsentationen skal vise alle elementerne i vektoren.

B.2 Liste

Den efterfølgende liste beskriver metoderne og egenskaberne i listekonceptet.

`__init__`: Dette er konstruktøren af instanser af listetypen, den skal kunne tage en samling af elementer, men også kunne acceptere ikke at få en samling. Kopier af elementerne i samlingen lægges i listen. Den forventede kompleksitet af denne metode er lineær i antallet af elementer i den givne samling.

`__del__`: Dette er destruktøren, den skal sørge for at frigive elementerne i listen, før listen selv frigives. Den forventede kompleksitet er lineær i antallet af elementer i listen.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen listen har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`swap`: Handler om at ombytte elementerne mellem to lister af samme implementering.

Den forventede kompleksitet er konstant, da det forventes at dette kun kræver ombytning af referencer.

`__len__`, `size`, `_get_size` og `resize`: Handler om antallet af elementer i listen.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af elementer, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af elementer i listen er konstant.

Metoden `resize` ændrer antallet af elementer til et givet antal. Metoden gives antallet af elementer, og værdien nye elementer skal have. Gives der ingen værdi er den `None`. Den forventede kompleksitet for metoden er lineær med antallet af elementer, der skal dannes eller fjernes fra listen.

Egenskaben `size` kan ikke bruges til at sætte antallet af elementer, grunden til dette er, at en egenskab kun kan tildeles en værdi, men ved sætning af antallet af elementer skal bruges både størrelsen og eventuelt et element der skal kopieres. Dette kunne implementeres, for eksempel kunne en bruger kombinere de to værdier til et par, som en metode så kunne adskille igen.

`back`, `_set_back`, `_get_back`, `pop_back` og `push_back`: Handler om at ændre det sidste element i listen.

Egenskaben `back` kalder metoderne `_get_back` og `_set_back` for henholdsvis at returnere og sætte det sidste element i listen.

Metoden `push_back` lægger et element tilsidst i listen og denne metode kaldes af `_set_back` når listen er tom. Metoden `pop_back` fjerner det sidste element fra listen.

Den forventede kompleksitet af metoderne er konstant.

`front`, `_set_front`, `_get_front`, `pop_front` og `push_front`: Handler om at ændre det første element i listen.

Egenskaben `front` kalder metoderne `_get_front` og `_set_front` for henholdsvis at returnere og sætte det første element i listen.

Metoden `push_front` lægger er element først i listen og denne metode kaldes af `_set_front` når listen er tom. Metoden `pop_front` fjerner det første element fra listen.

Den forventede kompleksitet af metoderne er konstant.

`clear`: Frigiver elementerne i listen, men frigiver ikke listen selv. Den forventede kompleksitet for dette er lineær i antallet af elementer i listen.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen listen har er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`__iter__`: Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første element i listen til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`, `_get_is_empty` **og** `__non_zero__`**:** Handler om at teste om listen er tom eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt hvis listen er tom, og metoden `__non_zero__` returnerer sandt hvis listen ikke er tom. Metoden er en del af Python og kunne udelades da Python tjekker på, hvad `__len__` returnerer, når metoden ikke findes. Den forventede kompleksitet af metoderne er konstant.

`insert`**:** Indsætter et eller flere elementer foran en iterator i listen. Den givne iterator skal være over listen. Det forventes, at listeimplemeringer bruger en ikke offentlig grænseflade til iteratoren. Den forventede kompleksitet af metoden er lineær i antallet af elementer, der skal indsættes.

`splice`**:** Indsætter elementerne fra en anden liste af samme implementering før en iterator i listen og tømmer den anden liste. Den givne iterator skal være over listen, det må forventes at listeimplementeringer bruger en ikke offentlig grænseflade til iteratoren. Den forventede kompleksitet af denne metode er konstant, da det antages, at eftersom begge lister er af samme implementering, at dette udelukkende er et spørgsmål om at rette i et konstant antal pegere.

`erase`**:** Sletter et eller flere elementer fra listen. Denne metode gives iteratorer til elementerne, der skal slettes fra listen. Det må forventes, at listeimplementeringer bruger en ikke offentlig grænseflade til iteratoren. Den forventede kompleksitet af denne metode er lineær i antallet af elementer, der skal slettes.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` **og** `__eq__`**:** Handler om at sammenligne to lister med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem elementerne i listerne i deres orden.

Metoderne implementerer sammenligningsoperatorerne for listeinstanser så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to lister. Den forventede kompleksitet er højest lineær i antallet af elementer i listen med færrest elementer.

`__repr__` **og** `__str__`**:** Handler om at få en tekstrepræsentation af listen.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af listens nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i listen, hvis repræsentationen skal vise alle elementerne i listen.

B.3 Toendetkø

Den efterfølgende liste beskriver metoderne og egenskaberne i toendetkøkonceptet.

`__init__`**:** Dette er konstruktøren af instanser af toendetkøtypen, den skal kunne tage en samling af elementer, men også kunne acceptere ikke at få en samling. Kopier af elementerne i samlingen lægges i den toendetkø. Den forventede kompleksitet af denne metode er lineær i antallet af elementer i den givne samling.

`__del__`**:** Dette er destruktøren, den skal sørge for at frigive elementerne i den toendetkø, før den toendetkø selv frigives. Den forventede kompleksitet er lineær i antallet af elementer i den toendetkø.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen den toendetkø har er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`swap`**:** Handler om at ombytte elementerne mellem to toendetkøer af samme implementering.

Den forventede kompleksitet er konstant, da det forventes, at dette kun kræver ombytning af et konstant antal referencer.

`__len__`**,** `size`**,** `_get_size` **og** `resize`**:** Handler om antallet af elementer i den toendetkø.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af elementer, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af elementer i den toendetkø er konstant.

Metoden `resize` ændrer antallet af elementer til et givet antal. Metoden gives antallet af elementer og værdi nye elementer skal have. Gives der ingen værdi er den `None`. Den forventede kompleksitet for metoden er lineær i antallet af elementer, der skal dannes eller fjernes fra den toendetkø.

Egenskaben `size` kan ikke bruges til at sætte antallet af elementer, grunden til dette er, at en egenskab kun kan tildeles en værdi, men ved sætning af antallet af elementer skal bruges både størrelsen og eventuelt et element, der skal kopieres. Dette kunne implementeres, for eksempel kunne en bruger kombinere de to værdier til et par, som en metode så kunne adskille igen.

`back`**,** `_set_back`**,** `_get_back`**,** `pop_back` **og** `push_back`**:** Handler om at ændre det sidste element i den toendetkø.

Egenskaben `back` kalder metoderne `_get_back` og `_set_back` for henholdsvis at returnere og sætte det sidste element i den toendetkø. Den forventede kompleksitet af metoderne er konstant.

Metoderne `push_back` og `pop_back` henholdsvis lægger og fjerner et element sidst i den toendetkø. Metoden `push_back` kaldes af `_set_back`, når den toendetkø er tom. Den forventede kompleksitet for metoderne er konstant.

`front`, `_set_front`, `_get_front`, `pop_front` og `push_front`: Handler om at ændre det første element i den toendetkø.

Egenskaben `front` kalder metoderne `_get_front` og `_set_front` for henholdsvis at returnere og sætte det første element i den toendetkø. Den forventede kompleksitet for metoderne er konstant.

Metoderne `push_front` og `pop_front` henholdsvis lægger og fjerner et element først i den toendetkø. Metoden `push_front` kaldes af `_set_front`, når den toendetkø er tom. Den forventede kompleksitet for metoderne er konstant.

`clear`: Frigiver elementerne i den toendetkø, men frigiver ikke den toendetkø selv. Den forventede kompleksitet for metoden er lineær i antallet af elementer i den toendetkø.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen den toendetkø har er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`__iter__`: Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første element i den toendetkø til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for metoden er konstant.

`is_empty`, `_get_is_empty` og `__non_zero__`: Handler om at teste om den toendetkø er tom eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt, hvis den toendetkø er tom, og metoden `__non_zero__` returnerer sandt, hvis den toendetkø ikke er tom Metoden `__non_zero__` er en del af Python og kunne udelades, da Python tjekker på, hvad `__len__` returnerer når metoden ikke findes. Den forventede kompleksitet af metoderne er konstant.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` og `__eq__`: Handler om at sammenligne to toendetkøer med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem elementerne i de toendetkøer i deres orden.

Metoderne implementerer sammenligningsoperatorerne for toendetkøinstanser, så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to toendetkøer. Den forventede kompleksitet er højest lineær i antallet af elementer i den toendetkø med færrest elementer.

`__repr__` og `__str__`: Handler om at få en tekstrepræsentation af den toendetkø.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af den toendetkøs nuværende tilstand, disse metoder er en del af Python og ikke af

specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i den toendetkø, hvis repræsentationen skal vise alle elementerne i den toendetkø.

B.4 Kø

Den efterfølgende liste beskriver metoderne og egenskaberne i køkonceptet.

`__init__`: Dette er konstruktøren af instanser af køtypen, den skal kunne tage en samling af elementer, men også kunne acceptere ikke at få en samling. Kopier af elementerne i samlingen lægges i køen. Den forventede kompleksitet af denne metode er lineær i antallet af elementer i den givne samling.

`__del__`: Dette er destruktøren, den skal sørge for at frigive elementerne i køen, før køen selv frigives. Den forventede kompleksitet er lineær i antallet af elementer i køen.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen køen har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`swap`: Handler om at ombytter elementerne mellem to køer af samme implementering.

Den forventede kompleksitet af metoden er konstant, da det forventes at dette kun kræver ombytning af et konstant antal referencer.

`__len__`, `size`, `_get_size` og `resize`: Handler om antallet af elementer i køen.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af elementer, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af elementer i køen er konstant.

Metoden `resize` ændrer antallet af elementer til et givet antal. Metoden gives antallet af elementer, og værdien nye elementer skal have. Gives der ingen værdi er den `None`. Den forventede kompleksitet for metoden er lineær i antallet af elementer, der skal dannes eller fjernes fra køen.

Egenskaben `size` kan ikke bruges til at sætte antallet af elementer, grunden til dette er, at en egenskab kun kan tildeles en værdi, men ved sætning af antallet af elementer skal bruges både størrelsen og eventuelt et element der skal kopieres. Dette kunne implementeres, for eksempel kunne en bruger kombinere de to værdier til et par, som en metode så kunne adskille igen.

`back`, `_set_back`, `_get_back` og `push_back`: Handler om at ændre det sidste element i køen.

Egenskaben `back` kalder metoderne `_get_back` og `_set_back` for henholdsvis at returnere og sætte det sidste element i køen. Metoden `push_back`

lægger er element sidst i køen og denne metode kaldes af `_set_back`, når køen er tom. Den forventede kompleksitet af metoderne er konstant.

`front`, `_set_front`, `_get_front` og `pop_front`: Handler om at ændre det første element i køen.

Egenskaben `front` kalder metoderne `_get_front` og `_set_front` for henholdsvis at returnere og sætte det første element i køen. Metoden `pop_front` fjerner det første element i køen. Den forventede kompleksitet for metoderne er konstant.

`clear`: Frigiver elementerne i køen, men frigiver ikke køen selv. Den forventede kompleksitet for dette er lineær i antallet af elementer i køen.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen køen har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`__iter__`: Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første element i køen til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`, `_get_is_empty` og `__non_zero__`: Handler om at teste om køen er tom eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt hvis køen er tom, og metoden `__non_zero__` returnerer sandt hvis køen ikke er tom. Metoden `__non_zero__` er en del af Python og kunne udelades, da Python tjekker på, hvad `__len__` returnerer når metoden ikke findes. Den forventede kompleksitet af metoderne er konstant.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` og `__eq__`: Handler om at sammenligne to køer med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem elementerne i køerne i deres orden.

Metoderne implementerer sammenligningsoperatorerne for køinstanser, så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to køer. Den forventede kompleksitet er højest lineær i antallet af elementer i køen med færrest elementer.

`__repr__` og `__str__`: Handler om at få en tekstrepræsentation af køen.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af køens nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i køen, hvis repræsentationen skal vise alle elementerne i køen.

B.5 Stak

Den efterfølgende liste beskriver metoderne og egenskaberne i stakkonceptet.

`__init__`**:** Dette er konstruktøren af instanser af staktypen, den skal kunne tage en samling af elementer, men også kunne acceptere ikke at få en samling. Kopier af elementerne i samlingen lægges i stakken. Den forventede kompleksitet af denne metode er lineær i antallet af elementer i den givet samling.

`__del__`**:** Dette er destruktøren, den skal sørge for at frigive elementerne i stakken, før stakken selv frigives. Den forventede kompleksitet er lineær i antallet af elementer i stakken.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen stakken har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`swap`**:** Handler om at ombytte elementerne mellem to stakke af samme implementering.

Den forventede kompleksitet af metoden konstant, da det forventes at dette kun kræver ombytning af et konstant antal referencer.

`__len__`**,** `size` **og** `_get_size`**:** Handler om antallet af elementer i stakken.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af elementer, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af elementer i stakken er konstant.

`top`**,** `_set_top`**,** `_get_top`**,** `push_top` **og** `pop_top`**:** Handler om at ændre det øverste element i stakken. Egenskaben `top` kalder metoderne `_get_top` og `_set_top` for henholdsvis at returnere og sætte det øverste element i stakken. Metoderne `push_top` og `pop_top` henholdvis lægger og fjerner et element øverst i stakken. Den forventede kompleksitet for metoderne er konstant.

`clear`**:** Frigiver elementerne i stakken, men frigiver ikke stakken selv. Den forventede kompleksitet for dette er lineær i antallet af elementer i stakken.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen stakken har er den eneste til elementet vil Python selv frigive elementet. (Garbage collector.)

`__iter__`**:** Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første element i stakken til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`, `_get_is_empty` **og** `__non_zero__`**:** Handler om at teste om stakken er tom eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt hvis stakken er tom, og metoden `__non_zero__` returnerer sandt hvis stakken ikke er tom, denne metode er en del af Python og kunne udelades da Python tjekker på, hvad `__len__` returnerer når metoden ikke findes. Den forventede kompleksitet af metoderne er konstant.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` **og** `__eq__`**:** Handler om at sammenligne to stakke med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem elementerne i stakkene i stakkenes orden.

Metoderne implementerer sammenligningsoperatorerne for stakinstanser, så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to stakke. Den forventede kompleksitet er højest lineær i antallet af elementer i stakken med færrest elementer.

`__repr__` **og** `__str__`**:** Handler om at få en tekstrepræsentation af stakken.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af stakkens nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i stakken, hvis repræsentationen skal vise alle elementerne i stakken.

B.6 Sæt

Den efterfølgende liste beskriver metoderne og egenskaberne i sætkonceptet. I de efterfølgende metoder nævnes nøgler, en nøgle er den værdi, der trækkes ud af et element og som sammenlignes med de andre nøgler i sættet.

`__init__`**:** Dette er konstruktøren af instanser af sættypen, den skal kunne tage en samling af elementer, en sammenligningsfunktion og en udtræksfunktion. For de to funktioner gælder det, at hvis de ikke gives, er sammenligningsfunktionen mindre end og udtræksfunktionen er hele elementet. Gives der ingen samling af elementer, bliver der ikke lagt nogen elementer i sættet i oprettelsen. Den forventede kompleksitet af denne metode er lineær i antallet af elementer i den givne samling.

`__del__`**:** Dette er destruktøren, den skal sørge for at frigive elementerne i sættet, før sættet selv frigives. Den forventede kompleksitet er lineær i antallet af elementer i sættet.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen sættet har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`swap`: Handler om at ombytte elementerne og de to funktioner mellem to sæt af samme implementering.

Den forventede kompleksitet er konstant, da det forventes, at dette kun kræver ombytning af et konstant antal referencer.

`__len__`, `size` og `_get_size`: Handler om antallet af elementer i sættet.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af elementer, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af elementer i sættet er konstant.

`clear`: Frigiver elementerne i sættet, men frigiver ikke sættet selv. Den forventede kompleksitet for dette er lineær i antallet af elementer i sættet.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen sættet har er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`__iter__`: Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første element i sættet til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`, `_get_is_empty` og `__non_zero__`: Handler om at teste om sættet er tomt eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt hvis sættet er tomt, og metoden `__non_zero__` returnerer sandt hvis sættet ikke er tomt. Denne metode er en del af Python og kunne udelades da Python tjekker på, hvad `__len__` returnerer, når metoden ikke findes. Den forventede kompleksitet af alle metoderne er konstant.

`comparator` og `_get_comparator`: Egenskaben `comparator` kalder metoden `_get_comparator`, der returnerer sammenligningsfunktionen, det er ikke tilsvarende muligt at sætte sammengligningsfunktionen.

Grunden til, at det ikke er muligt at sætte funktionen er, at den nye funktion kan ændre på ordenen mellem nøglerne i sættet, og dermed ville forårsage, at hele sættet skulle omordnes. Den anden grund er, at en ny sammenligningsfunktion kunne forårsage at elementer i sættet får samme nøgle, og da nøgler skal være unikke skulle et af elementerne fjernes. Den forventede kompleksitet for dette er konstant.

`extractor` og `_get_extractor`: Egenskaben `extractor` kalder metoden `_get_extractor`, der returnerer udtræksfunktionen, Det er ikke tilsvarende muligt at sætte udtræksfunktionen.

Grunden til, at det ikke er muligt at sætte funktionen er, at den nye funktion ændrer på hvad der en nøgle i et element og dermed også kan ændre på ordenen mellem elemternes nøgler og dermed skal sættet omordnes.

Den anden grund er, at en ny udtræksfunktion kunne forårsage at elementer i sættet får samme nøgle, og da nøgler skal være unikke skulle et af elementerne fjernes. Den forventede kompleksitet for dette er konstant.

`lower_bound`, `upper_bound` **og** `equal_range`**:** Metoden `lower_bound` tager en nøgle og returnerer iteratoren til første element i sættet, hvis nøglen ikke er før den givne i sættets orden.

Metoden `upper_bound` returnerer iteratoren til det første element, hvis nøglen er efter den givne nøgle i sættets orden.

Metoden `equal_range` returnerer iteratorerne returnerede fra `lower_bound` og `upper_bound` givet nøglen, som metoden kaldes med. Den returnerede iterator vil være ved sættets ende, hvis alle elementernes nøgle var før den givne i søgningen.

Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`find`**:** Metoden `find` tager en nøgle og returnerer iteratoren til elementet i sættet, hvis nøgle er den givne, eller returnerer iteratoren forbi det sidste element i sættet, hvis intet element har den givne nøgle. Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`insert`**:** Metoden `insert` kan indsætte et element eller en samling af elementer. Metoden kan tage en iterator i sættet. Tanken er, at givet en iterator der er før de elementer, der indsættes, kan indsætning gøres hurtigere ved at mindre skal gennemsøges. Dette forudsætter at implementeringen kan dette. Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`erase`**:** Metoden `erase` sletter alle elementerne i sættet, der har en given nøgle, peget på af en iterator eller mellem to iteratorer. Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`count`**:** Metoden `count` tager en nøgle og returnerer antallet af gange denne nøgle findes i sættet, dette kan højest være 1. (Nøgler er unikke i sættet.) Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` **og** `__eq__`**:** Handler om at sammenligne to sæt med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem elementerne i sættene i deres orden.

Metoderne implementerer sammenligningsoperatorerne for sætinstanser, så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to sæt. Den forventede kompleksitet er højest lineær i antallet af elementer i sættet med færrest elementer.

Sammenligning, der foretages er mellem nøglerne i sættene.

`__repr__` **og** `__str__`**:** Handler om at få en tekstrepræsentation af sættet.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af sættets nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i sættet, hvis repræsentationen skal vise alle elementerne i sættet.

B.7 Multisæt

Den efterfølgende liste beskriver metoderne og egenskaberne i multisætkonceptet. I de efterfølgende metoder nævnes nøgler, en nøgle er den værdi der trækkes ud af et element og som sammenlignes med de andre nøgler i multisættet.

`__init__`: Dette er konstruktøren af instanser af multisættypen, den skal kunne tage en samling af elementer, en sammenligningsfunktion og en udtræksfunktion. For de to funktioner gælder det, at hvis de ikke gives er sammenligningsfunktionen mindre end og udtræksfunktionen er hele elementet. Gives der ingen samling af elementer bliver, der ikke lagt nogen elementer i multisættet i oprettelsen. Den forventede kompleksitet af denne metode er lineær i antallet af elementer i den givne samling.

`__del__`: Dette er destruktøren, den skal sørge for at frigive elementerne i multisættet, før multisættet selv frigives. Den forventede kompleksitet er lineær i antallet af elementer i multisættet.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen multisættet har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`swap`: Handler om at ombytter elementerne og funktionerne mellem to multisæt af samme implementering.

Den forventede kompleksitet er konstant, da det forventes at dette kun kræver ombytning af et konstant antal referencer.

`__len__`, `size` og `_get_size`: Handler om antallet af elementer i multisættet.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af elementer, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af elementer i multisættet er konstant.

`clear`: Frigiver elementerne i multisættet, men frigiver ikke multisættet selv. Den forventede kompleksitet for dette er lineær i antallet af elementer i multisættet.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen multisættet har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`__iter__`: Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første element i multisættet til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`, `_get_is_empty` og `__non_zero__`: Handler om at teste om multisættet er tomt eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt, hvis multisættet er tomt, og metoden `__non_zero__` returnerer sandt, hvis multisættet ikke er tomt, denne metode er en del af Python og kunne udelades da Python tjekker på, hvad `__len__` returnerer, når metoden ikke findes. Den forventede kompleksitet af alle metoderne er konstant.

`comparator` og `_get_comparator`: Egenskaben `comparator` kalder metoden `_get_comparator`, der returnerer sammenligningsfunktionen, det er ikke tilsvarende muligt at sætte sammenligningsfunktionen.

Grunden til, at det ikke er muligt at sætte funktionen er, at den nye funktion kan ændre på ordenen mellem nøglerne i multisættet, og dermed ville forårsage at hele multisættet skulle omordnes. Den forventede kompleksitet for dette er konstant.

`extractor` og `_get_extractor`: Egenskaben `extractor` kalder metoden `_get_extractor`, der returnerer udtræksfunktionen, Det er ikke tilsvarende muligt at sætte udtræksfunktionen.

Grunden til, at det ikke er muligt at sætte funktionen er, at den nye funktion ændrer på hvad der en nøgle i et elementet, og dermed også kan ændre på ordenen mellem elementernes nøgler og dermed skal multisættet omordnes. Den forventede kompleksitet for dette er konstant.

`lower_bound`, `upper_bound` og `equal_range`: Metoden `lower_bound` tager en nøgle og returnerer iteratoren til første element i multisættet, hvis nøgle ikke er før den givne i multisættets orden.

Metoden `upper_bound` returnerer iteratoren til det første element, hvis nøgle er efter den givne nøgle i multisættets orden.

Metoden `equal_range` returnerer iteratorerne returnerede fra `lower_bound` og `upper_bound` givet nøglen som metoden kaldes med. De returnerede iteratorer vil være ved multisættets ende, hvis alle elementernes nøgle var før den givne i søgningen.

Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`find`: Metoden `find` tager en nøgle og returnerer iteratoren til elementet i multisættet, hvis nøgle er den givne, eller iteratoren forbi det sidste element i multisættet, hvis intet element har den givne nøgle. Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`insert`: Metoden `insert` kan indsætte et element eller en samling af elementer. Metoden kan tage en iterator i multisættet. Tanken er, at givet en iterator der er før de elementer, der indsættes, kan indsætning gøres hurtigere ved at mindre skal gennemsøges, dette forudsætter at implementeringen kan dette. Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`erase`: Metoden `erase` sletter alle elementerne i multisættet, der har en given nøgle, peget på af en iterator eller mellem to iteratorer. Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`count`: Metoden `count` tager en nøgle og returnerer antallet af gange denne nøgle findes i multisættet. Den forventede kompleksitet er $\lg n$, hvor n er antallet af elementer.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` og `__eq__`: Handler om at sammenligne to multisæt med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem elementerne i multisættene i deres orden.

Metoderne implementerer sammenligningsoperatorerne for multisætinstanser, så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to multisæt. Den forventede kompleksitet er højest lineær i antallet af elementer i multisættet med færrest elementer.

Sammenligning, der foretages, er ikke mellem elementerne i multisættene, men mellem udtrækket af elementerne i multisættene.

`__repr__` og `__str__`: Handler om at få en tekstrepræsentation af multisættet.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af sættets nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i multisættet, hvis repræsentationen skal vise alle elementerne i multisættet.

B.8 Kort

Den efterfølgende liste beskriver metoderne og egenskaberne i kortkonceptet.

I denne beskrivelse er betegnelsen par brugt, dette dækker over kombinationen af en nøgle og et element, disse er i et kort to separate datastrukturer, der gives som et par.

`__init__`: Dette er konstruktøren af instanser af korttypen, den skal kunne tage en samling af par og en sammenligningsfunktion. For sammenligningsfunktionen gælder det, at hvis den ikke gives, er sammenligningen mindre end. Gives der ingen samling af par bliver, der ikke lagt nogen par i kortet i oprettelsen. Den forventede kompleksitet af denne metode er lineær i antallet af par i den givne samling.

`__del__`: Dette er destruktøren. Den skal sørge for at frigive parrene i kortet, før kortet selv frigives. Den forventede kompleksitet er lineær i antallet af par i kortet.

At frigive parrene udføres ved at fjerne referencen til dem, hvis referencen kortet har, er den eneste til parret, vil Python selv frigive parret. (Garbage collector.)

`swap`: Handler om at ombytte parrene og sammenligningsfunktionen mellem to kort af samme implementering.

Den forventede kompleksitet er konstant, da det forventes at dette kun kræver ombytning af et konstant antal referencer.

`__len__`, `size` og `_get_size`: Handler om antallet af par i kortet.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af par, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af par i kortet er konstant.

`clear`: Frigiver parrene i kortet, men frigiver ikke kortet selv. Den forventede kompleksitet for dette er lineær i antallet af par i kortet.

At frigive parrene udføres ved at fjerne referencen til dem, hvis referencen køen har, er den eneste til parret, vil Python selv frigive parret. (Garbage collector.)

`__iter__`: Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første par i kortet til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`, `_get_is_empty` og `__non_zero__`: Handler om at teste om kortet er tomt eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt, hvis kortet er tomt, og metoden `__non_zero__` returnerer sandt, hvis kortet ikke er tomt. Denne metode er en del af Python og kunne udelades da Python tjekker på, hvad `__len__` returnerer, når metoden ikke findes. Den forventede kompleksitet af alle metoderne er konstant.

`comparator` og `_get_comparator`: Egenskaben `comparator` kalder metoden `_get_comparator`, der returnerer sammenligningsfunktionen, det er ikke tilsvarende muligt at sætte sammengligningsfunktionen.

Grunden til, at det ikke er muligt at sætte funktionen er, at den nye funktion kan ændre på ordenen mellem nøglerne, på parrene der er i kortet, og dermed ville forårsage at kortet skal omordnes.

`lower_bound`, `upper_bound` og `equal_range`: Metoden `lower_bound` tager en nøgle og returnerer iteratoren til det første par i kortet, hvis nøgle ikke er før den givne i kortets orden.

Metoden `upper_bound` returnerer iteratoren til det første element, hvis nøgle er efter den givne nøgle i kortets orden.

Metoden `equal_range` returnerer iteratorerne returnerede fra `lower_bound` og `upper_bound` givet nøglen, som metoden kaldes med. De returnerede iteratorer vil være ved kortets ende, hvis alle nøgler i parrene var før den givne i søgningen.

Den forventede kompleksitet for metoderne er $\lg n$, hvor n er antallet af par.

`find`: Metoden `find` tager en nøgle og returnere iteratoren til parret i kortet, hvis nøgle er den givne, eller iteratoren forbi det sidste par i kortet, hvis intet par har den givne nøgle. Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`insert`: Metoden `insert` kan indsætte et par eller en samling af par. Metoden kan tage en iterator i kortet. Yanken er at givet en iterator der peger på et par før parrene, der indsættes, kan indsætning gøres hurtigere ved at mindre skal gennemsøges, dette forudsætter at implementeringen kan dette. Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`erase`: Metoden `erase` sletter alle par, hvis nøgle er lig med en givet nøgle, peget på af en iterator eller mellem to iteratorer. Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`count`: Metoden `count` tager en nøgle og returnerer antallet af gange denne nøgle findes i kortet, dette kan højest være en da nøgler er unikke. Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` og `__eq__`: Handler om at sammenligne to kort med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem parrene i kortene i deres orden.

Metoderne implementerer sammenligningsoperatorerne for kortinstanser, så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to kort. Den forventede kompleksitet er højest lineær i antallet af elementer i kortet med færrest elementer.

Sammenligningen, der foretages, er mellem nøglerne i parrene.

`__repr__` og `__str__`: Handler om at få en tekstrepræsentation af kortet.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af kortets nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i kortet, hvis repræsentationen skal vise alle elementerne i kortet.

B.9 Multikort

Den efterfølgende liste beskriver metoderne og egenskaberne i multikortkonceptet.

I denne beskrivelse er betegnelsen par brugt, dette dækker over kombinationen af en nøgle og et element, disse er i et multikort to separate datastrukturer, der gives som et par.

`__init__`: Dette er konstruktøren af instanser af multikorttypen, den skal kunne tage en samling af par og en sammenligningsfunktion. For sammenligningsfunktionen gælder det, at hvis den ikke gives er sammenligningen mindre end. Gives der ingen samling af par, bliver der ikke lagt nogen par i multikortet i oprettelsen. Den forventede kompleksitet af denne metode er lineær i antallet af par i den givne samling.

`__del__`: Dette er destruktøren, den skal sørge for at frigive parrene i multikortet, før multikortet selv frigives. Den forventede kompleksitet er lineær i antallet af par i multikortet.

At frigive parrene udføres ved at fjerne referencen til dem, hvis referencen multikortet har, er den eneste til parret, vil Python selv frigive parret. (Garbage collector.)

`swap`: Handler om at ombytte parrene og sammenligningsfunktionen mellem to multikort af samme implementering.

Den forventede kompleksitet er konstant, da det forventes at dette kun kræver ombytning af et konstant antal referencer.

`__len__`, `size` og `_get_size`: Handler om antallet af par i multikortet.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af par, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af par i multikortet er konstant.

`clear`: Frigiver parrene i multikortet, men frigiver ikke multikortet selv. Den forventede kompleksitet for dette er lineær i antallet af par i multikortet.

At frigive parrene udføres ved at fjerne referencen til dem, hvis referencen multikortet har, er den eneste til parret, vil Python selv frigive elementet. (Garbage collector.)

`__iter__`: Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første par i multikortet til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`, `_get_is_empty` og `__non_zero__`: Handler om at teste om multikortet er tomt eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt, hvis multikortet er tomt, og metoden `__non_zero__` returnerer sandt, hvis multikortet ikke er tomt, denne metode er en del af Python og kunne udelades da Python tjekker på, hvad `__len__` returnerer, når metoden ikke findes. Den forventede kompleksitet af metoderne er konstant.

`comparator` og `_get_comparator`: Egenskaben `comparator` kalder metoden `_get_comparator`, der returnerer sammenligningsfunktionen, det er ikke tilsvarende muligt at sætte sammenligningsfunktionen.

Grunden til, at det ikke er muligt at sætte funktionen er, at den nye funktion kan ændre på ordenen mellem nøglerne, på parrene, der er i multikortet, og dermed ville forårsage at multikortet skal omordnes.

`lower_bound`, `upper_bound` og `equal_range`: Metoden `lower_bound` tager en nøgle og returnerer iteratoren til første par i multikortet, hvis nøgle ikke er før den givne i multikortets orden.

Metoden `upper_bound` returnerer iteratoren til det første par, hvis nøgle er efter den givne nøgle i multikortets orden.

Metoden `equal_range` returnerer iteratorerne returnerede fra `lower_bound` og `upper_bound` givet nøglen som metoden kaldes med. De returnerede iterator vil være ved multikortets ende, hvis alle nøglerne i parrene var før den givne i søgningen.

Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`find`: Metoden `find` tager en nøgle og returnerer iteratoren til parret i multikortet, hvis nøgle er den givne, eller iteratoren forbi det sidste par i multikortet, hvis intet par har den givet nøgle. Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`insert`: Metoden `insert` kan indsætte et par eller en samling af par. Metoden kan tage en iterator i multikortet. Tanken er at givet en iterator der er før de par der indsættes kan indsætning gøres hurtigere ved at mindre skal gennemsøges, dette forudsætter at implementeringen kan dette. Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`erase`: Metoden `erase` sletter alle par, hvis nøgle er lig med en given nøgle, peget på af en iterator eller mellem to iteratorer. Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`count`: Metoden `count` tager en nøgle og returnerer antallet af gange denne nøgle findes i multikortet. Den forventede kompleksitet er $\lg n$, hvor n er antallet af par.

`__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__` og `__eq__`: Handler om at sammenligne to multikort med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem parrene i multikortnee i deres orden.

Metoderne implementerer sammenligningsoperatorerne for multikortinstanser, så operatorerne <, <=, >, >=, == og != kan bruges mellem to multikort. Den forventede kompleksitet er højest lineær i antallet af par i multisættet med færrest par.

Sammenligningen, der foretages, er mellem nøgledelen af parrene i multikortene.

`__repr__` og `__str__`: Handler om at få en tekstrepræsentation af multikortet.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af multikortets nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af par i multikortet, hvis repræsentationen skal vise alle parrene i multikortet.

B.10 Prioritetskø

Den efterfølgende liste beskriver metoderne og egenskaberne i prioritetskøkonceptet.

`__init__`: Dette er konstruktøren af instanser af prioritetskøtypen, den skal kunne tage en samling af elementer, en sammenligningsfunktion og en udtræksfunktion. For de to funktioner gælder det, at hvis de ikke gives er sammenligningsfunktionen mindre end og udtræksfunktionen er hele elementet. Gives der ingen samling af elementer bliver der ikke lagt nogen elementer i prioritetskøen i oprettelsen. Den forventede kompleksitet af denne metode er lineær i antallet af elementer i den givne samling.

`__del__`: Dette er destruktøren, den skal sørge for at frigive elementerne i prioritetskøen, før prioritetskøen selv frigives. Den forventede kompleksitet er lineær i antallet af elementer i prioritetskøen.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen prioritetskøen har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`swap`: Handler om at ombytte elementerne og sammenligningsfunktionen mellem to prioritetskøer af sammen implementering.

Den forventede kompleksitet er konstant, da det forventes at dette kun kræver ombytning af et konstant antal referencer.

`__len__`, `size` og `_get_size`: Handler om antallet af elementer i prioritetskøen.

Egenskaben `size` og metoden `__len__` returnerer begge antallet af elementer, `size` kalder metoden `_get_size`. Den forventede kompleksitet for at få antallet af elementer i prioritetskøen er konstant.

`clear`**:** Frigiver elementerne i prioritetskøen, men frigiver ikke prioritetskøen selv. Den forventede kompleksitet for dette er lineær i antallet af elementer i prioritetskøen.

At frigive elementerne udføres ved at fjerne referencen til dem, hvis referencen prioritetskøen har, er den eneste til elementet, vil Python selv frigive elementet. (Garbage collector.)

`__iter__`**:** Metoden `__iter__` er ikke en del af specifikationen, men en del af Python. Metoden returnerer en iterator, der kan iterere fra det første element i prioritetskøen til det sidste. Dette kan dermed ses som en substitut for metoderne `begin` og `end` fra specifikationen. Den forventede kompleksitet for dette er konstant.

`is_empty`**,** `_get_is_empty` **og** `__non_zero__`**:** Handler om at teste om prioritetskøen er tom eller ej.

Egenskaben `is_empty` kalder metoden `_get_is_empty` og returnerer sandt hvis prioritetskøen er tom, og metoden `__non_zero__` returnerer sandt hvis prioritetskøen ikke er tom, denne metode er en del af Python og kunne udelades da Python tjekker på, hvad `__len__` returnerer, når metoden ikke findes. Den forventede kompleksitet af metoderne er konstant.

`comparator` **og** `_get_comparator`**:** Egenskaben `comparator` kalder metoden `_get_comparator`, der returnerer sammenligningsfunktionen, Det er ikke tilsvarende muligt at sætte sammenligningsfunktionen.

Grunden til, at funktionen ikke kan sættes er, at den nye funktion kan ændre på ordenen mellem elementerne i prioritetskøen, og dermed ville forårsage at prioritetskøen skulle omordnes.

`extractor` **og** `_get_extractor`**:** Egenskaben `extractor` kalder metoden `_get_extractor`, der returnerer udtræksfunktionen, det er ikke tilsvarende muligt at sætte udtræksfunktionen.

Grunden til, at funktionen ikke kan sættes er, at den nye funktion kan ændre på ordenen mellem elementerne i prioritetskøen, og dermed ville forårsage at prioritetskøen skal omordnes.

`pop`**,** `top` **og** `_get_top`**:** Egenskaben `top` kalder metoden `_get_top` der returnerer det første element i prioritetskøen og metoden `pop` fjerner det første element fra prioritetskøen.

`push`**:** Metoden `push` tager et element og indsætter det i prioritetskøen efter elementets prioritet.

`__lt__`**,** `__le__`**,** `__gt__`**,** `__ge__`**,** `__ne__` **og** `__eq__`**:** Handler om at sammenligne to prioritetskøer med hinanden.

Sammenligningen er lexiografisk, hvilket vil sige, at den ønskede sammenligning foretages mellem elementerne i prioritetskøerne i deres orden.

Metoderne implementerer sammenligningsoperatorerne for prioritetskø-instanser, så operatorerne `<`, `<=`, `>`, `>=`, `==` og `!=` kan bruges mellem to prioritetskøer. Den forventede kompleksitet er højest lineær i antallet af elementer i prioritetskøen med færrest par.

Sammenligningen, der foretages, er mellem udtræksdelen af elementerne i prioritetskøen

`__repr__` og `__str__`: Handler om at få en tekstrepræsentation af priotetskøen.

Metoderne `__repr__` og `__str__` returnerer en tekstrepræsentation af prioritetskøens nuværende tilstand, disse metoder er en del af Python og ikke af specifikationen. Den forventede kompleksitet er højest lineær i antallet af elementer i prioritetskøen, hvis repræsentationen skal vise alle elementerne i prioritetskøen.

Bilag C

Dokumentation

Dette bilag indeholder formatskabeleoner og standard sætninger brugt i CPH collections. Formatterne er tilgængelige i dokumentationen af biblioteket.

C.1 Format

Formatter brugt i skrivningen af dokumentationsstrenge i kildekodefilerne i CPH collections.

Listing C.1: Dokumentationsformattet for klasser

```
"""
DESCRIPTION

Examples
========
SHORT_DESCRIPTION. ::

  EXAMPLE

"""
```

Alt skrevet med store bogstaver skal individuliseres for klassen. Hvis en klasse har flere eksempler gentages konstruktionen efter eksempler overskriften.

Listing C.2: Dokumentationsformattet for abstrakte klasser

```
"""
DESCRIPTION

Inheriting from this class the following must be implemented:

  - *Methods*

    - *METHOD_NAME*

  - *Properties*

    - *PROPERTY_NAME*

"""
```

Alt skrevet med store bogstaver skal individuliseres for den abstakte klasse. Hvis en klasse har flere abstrakte metoder eller egenskaber gentages deres linier for, hver egenskab eller egenskab. Dette skaber en punktliste i den genererede dokumentation.

Listing C.3: Dokumentationsformattet for funktioner og metoder

```
1       """
2       ONE_LINE_DESCRIPTION
3
4       LONGER_DESCRIPTION
5
6       Parameters
7       ==========
8       NAME : TYPE, OPTIONAL
9         SHORT_DESCRIPTION IF_OPTIONAL
10
11      Returns
12      =======
13      TYPE
14        SHORT_DESCRIPTION
15
16      Raises
17      ======
18      TYPE
19         SHORT_DESCRIPTION
20
21      Examples
22      ========
23      SHORT_DESCRIPTION. ::
24
25        EXAMPLE
26      """
```

Alt skrevet med store bogstaver skal individuliseres for funktionen eller metoden.

Hvis en funktion eller metode har tager flere argumenter gentages konstruktionen efter parametrer overskriften. Tager den ingen argumenter skrives der `Node`.

Hvis en funktion eller metode har returnerer en værdi skrives typen eventuelt med en kort beskrivelse. Returnerer den ingen værdi skrives der `Node`.

Hvis en funktion eller metode har rejser undtagelser gentages konstruktionen efter undtagelses overskriften. Rejses der ingen undtagelser skrives der `Node`.

Hvis en funktion eller metode har flere eksempler gentages konstruktionen efter eksempler overskriften.

Listing C.4: Dokumentationsformattet for egenskaber

```
1       """
2       Property :
3         **Read** : :py:func:`FUNCTION_NAME`
4
5         **Write** : :py:func:`FUNCTION_NAME`
6       """
```

Alt skrevet med store bogstaver skal individuliseres for egenskaben.

funktionen egenskaben kalder skrives, hvor `FUNCTION_NAME` står. Kan en egenskab ikke læses eller skrives til slettes den pågældende linie.

C.2 Sætninger

Standard sætninger brugt i skrivningen af dokumentationen.

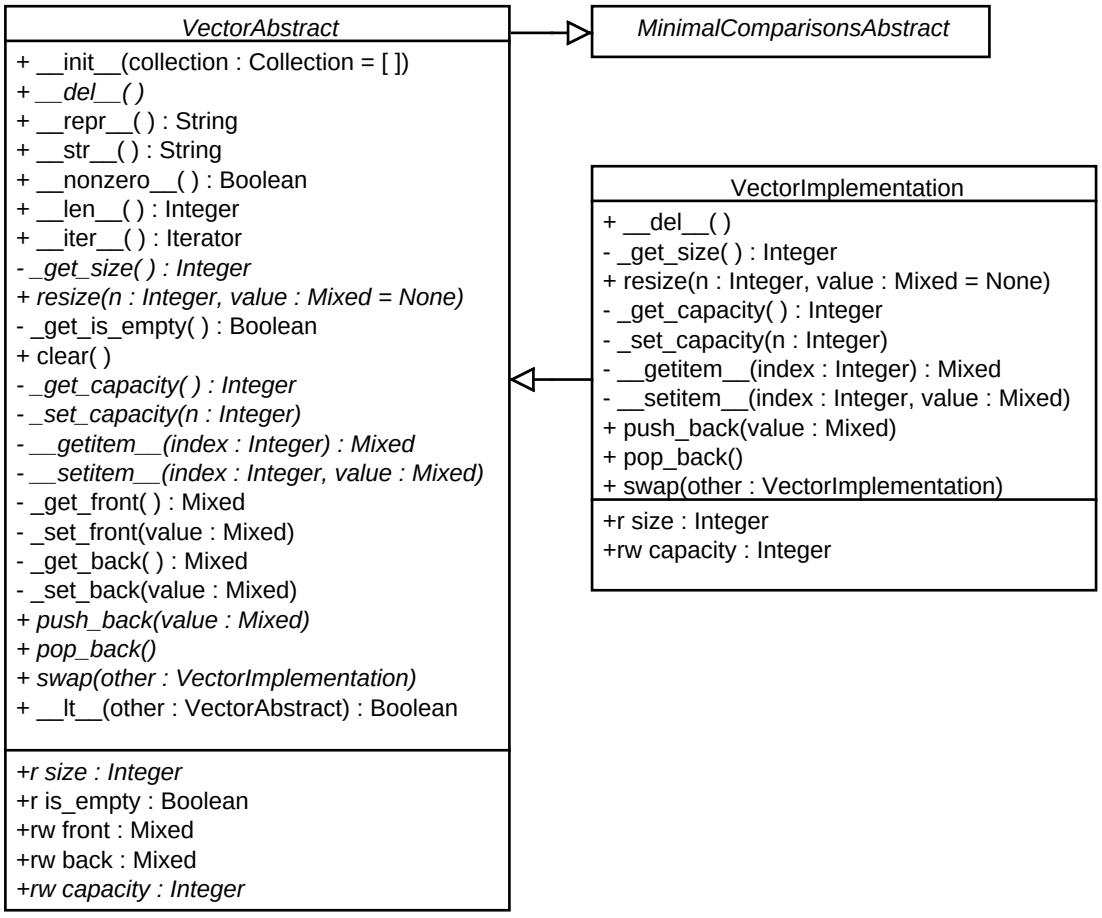
```
1
2  Creating objects sentences
3  ==========================
4  Create a TYPENAME object.
5  Create two TYPENAME objects with N and M elements.
6  Create a TYPENAME object with N elements and get its PROPERTYNAME property.
7  Create a TYPENAMEE object with N elements and set its PROPERTYNAME property.
8  Create a TYPENAME object with no elements and push two elements on it.
9  Create a TYPENAME object with N elements and clear it.
10 Create a TYPENAME object and call *del* on it.
11
12 Delete object sentences
13 ======================
14 Delete theTYPENAME object.
15
16
17 Default sentences
18 ================
19 If not given it defaults to an empty list.
20 If not given it defaults to *None*.
21
22
23 Abstract sentences
24 ==================
25 This is the constructor of an abstract class, so it can't␣be␣called␣directly.
26
27
28 Comparison␣sentences
29 ====================
30 The␣TYPENAME␣to␣compare␣with.
31 Compare␣the␣two␣objects␣in␣a␣<␣comparison.
32 Is␣called␣when␣the␣TYPENAME␣object␣is␣the␣1st␣operant␣in␣the␣statement␣<,␣>,␣
       ==,␣!=,␣<=␣and␣>=
33
34 Return␣sentences
35 ================
36 Return␣*VALUE*;␣if␣the␣TYPENAME␣has␣less␣elements␣than␣the␣given␣TYPENAME.
37 Return␣the␣number␣of␣elements␣in␣the␣TYPENAME.
38 Return␣*VALUE*;␣if␣the␣TYPENAME␣is␣lexicographical␣less␣than␣the␣given␣
       TYPENAME.
39 Return␣a␣string␣representation␣of␣the␣TYPENAME.
40 Return␣the␣TYPENAME␣converted␣to␣a␣string.
41
42 Swap␣sentences
43 ==============
44 Swap␣the␣elements␣of␣two␣TYPENAMEs.
45 The␣TYPENAME␣to␣swap␣with.
46
47
48 Conditional␣sentences
49 =====================
50 If␣**other**␣is␣not␣an␣TYPE.
51
52 Abstractbaseclass␣constructor␣examples
53 ======================================
54 This␣is␣the␣constructor␣of␣an␣abstract␣class␣and␣should␣not␣be␣called␣
       directly␣so␣no␣examples␣are␣provided.
```

Bilag D

UML-diagrammer for abstrakte superklasser af beholdertyper

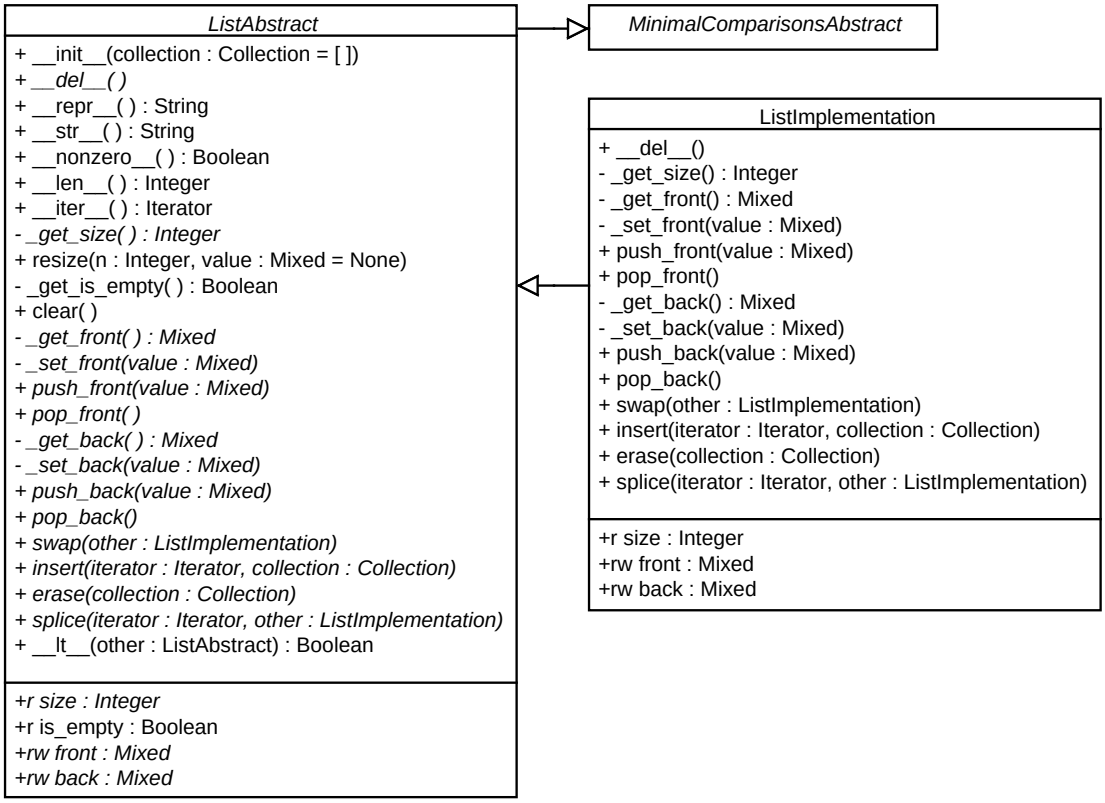
Dette bilag indeholder klassediagrammerne for de abstrakte superklasser for beholdertyperne i CPH collections. Det er kun de abstrakte klasser, der findes i CPH collections, mens klasserne vist som implementerings klasser kun er der, for at vise hvilke metoder en implementerings klasse skal implementere.

D.1 Vektor



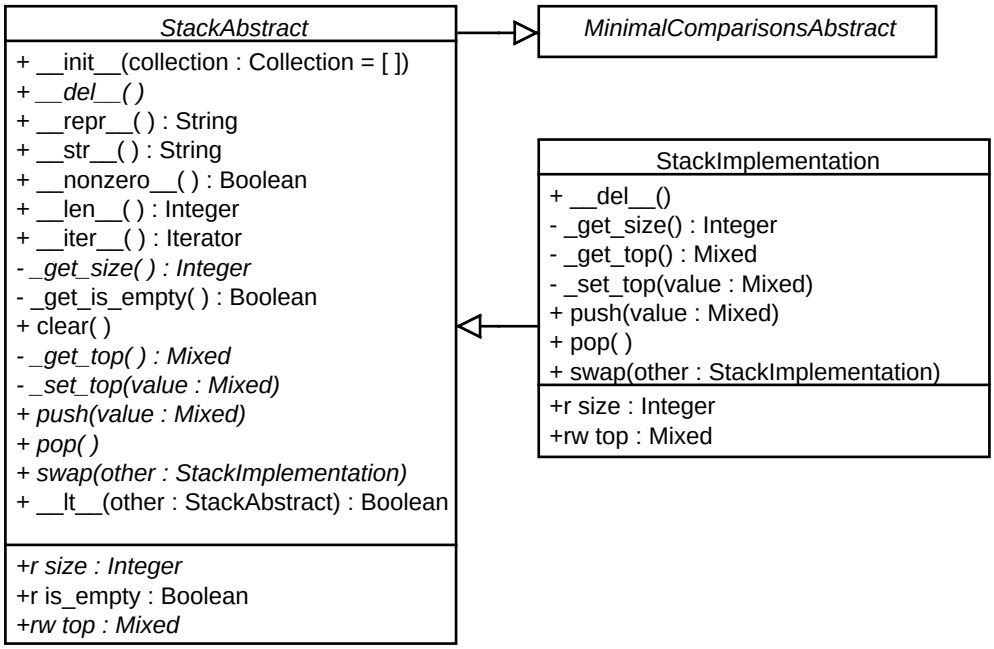
Figur D.1: Klassediagrammet for den abstrakte superklasse for vektor implementeringer.

D.2 Liste



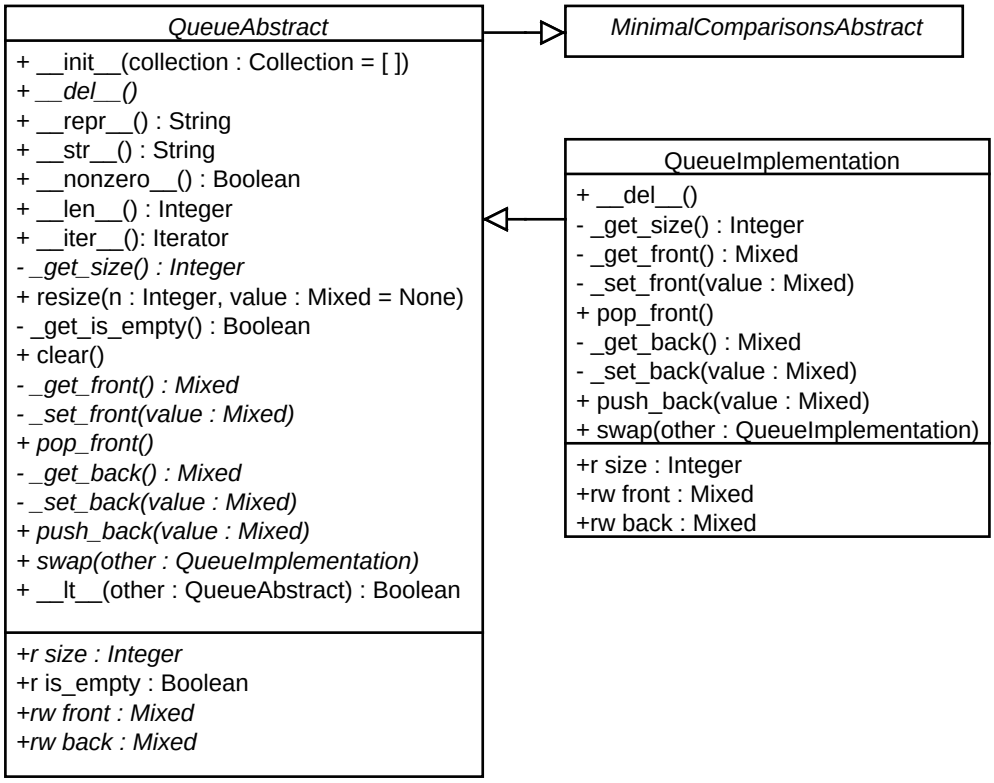
Figur D.2: Klassediagrammet for den abstrakte superklasse for liste implementeringer.

D.3 Stak



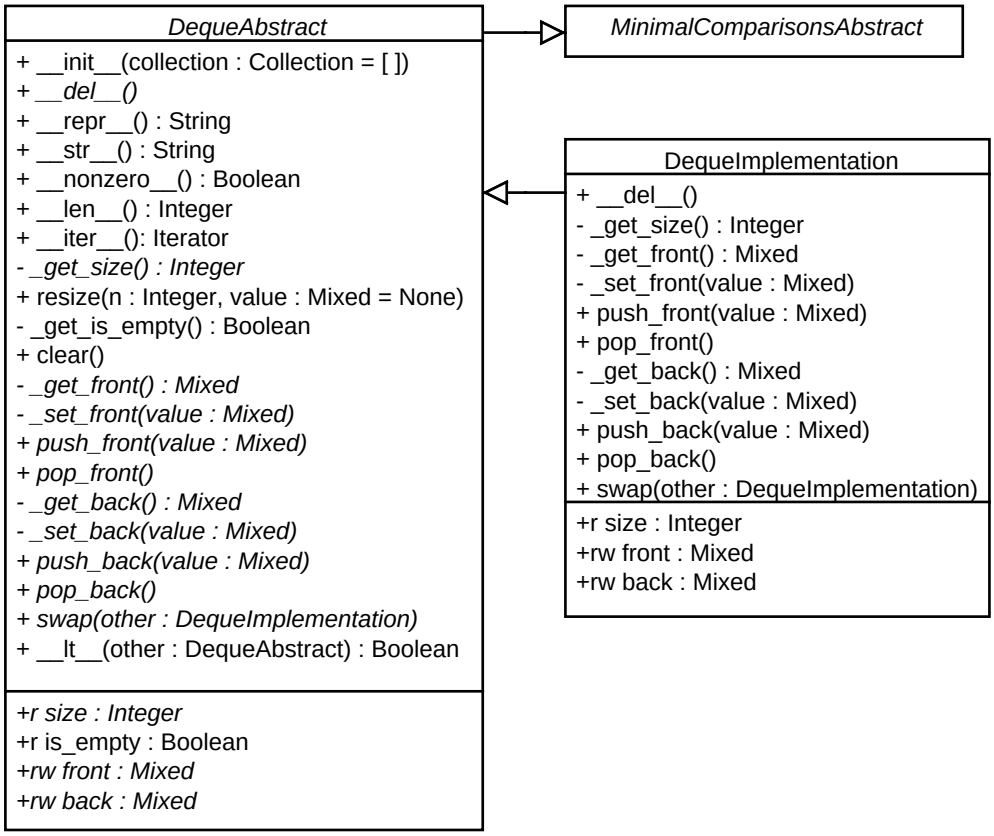
Figur D.3: Klassediagrammet for den abstrakte superklasse for stak implementeringer.

D.4 Kø



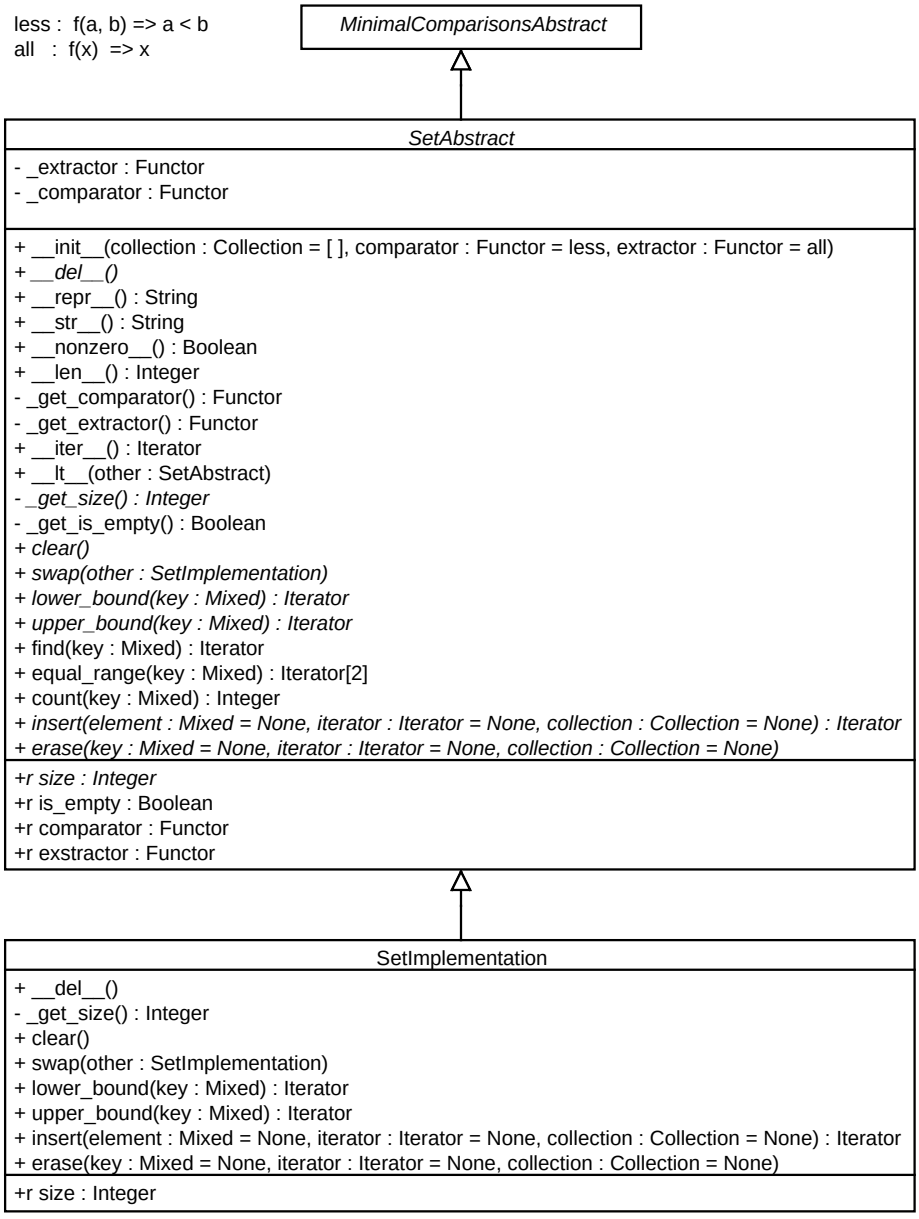
Figur D.4: Klassediagrammet for den abstraktebaseklasse for kø implementeringer.

D.5 Toendetkø



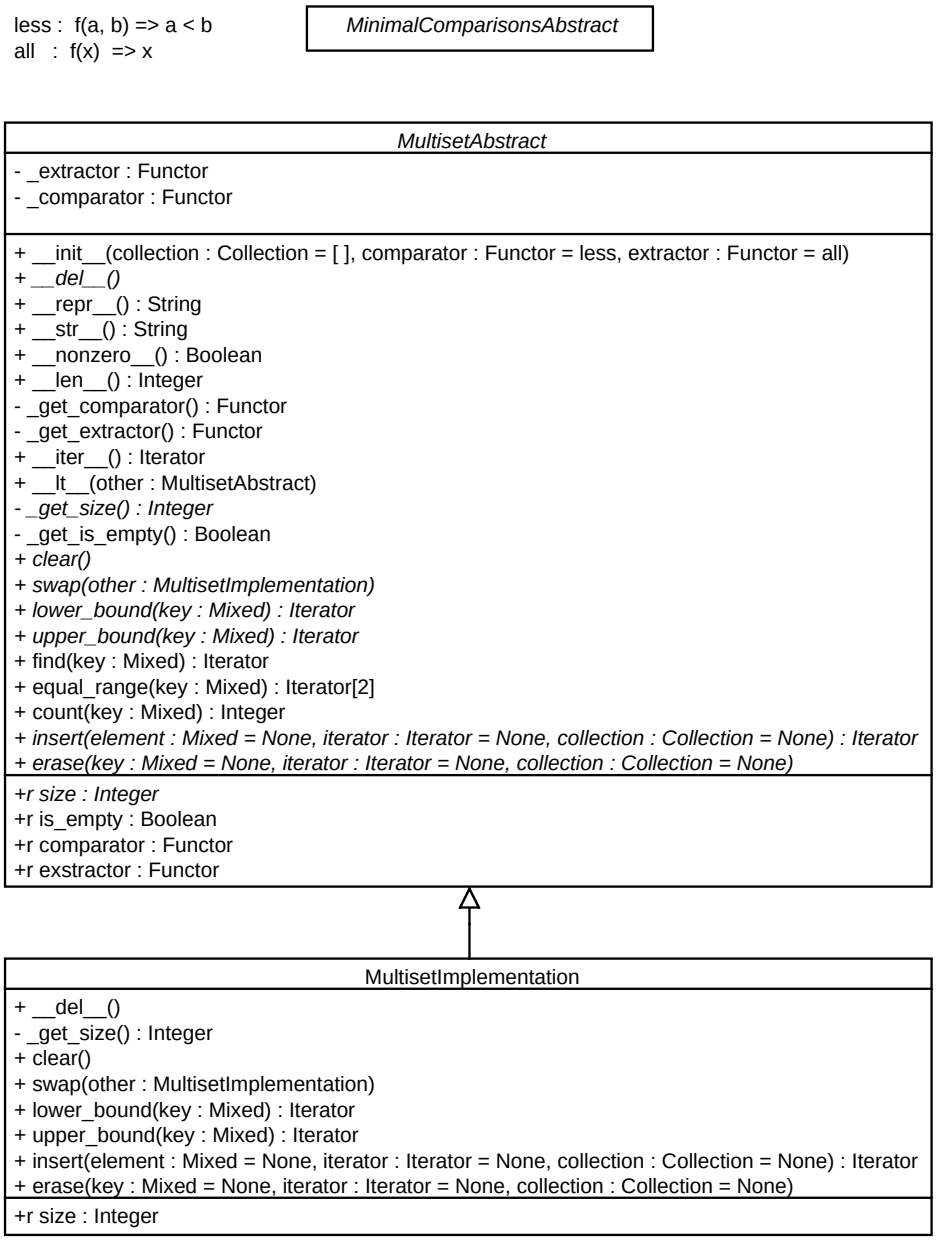
Figur D.5: Klassediagrammet for den abstrakte superklasse for toendetkø implementeringer.

D.6 Sæt



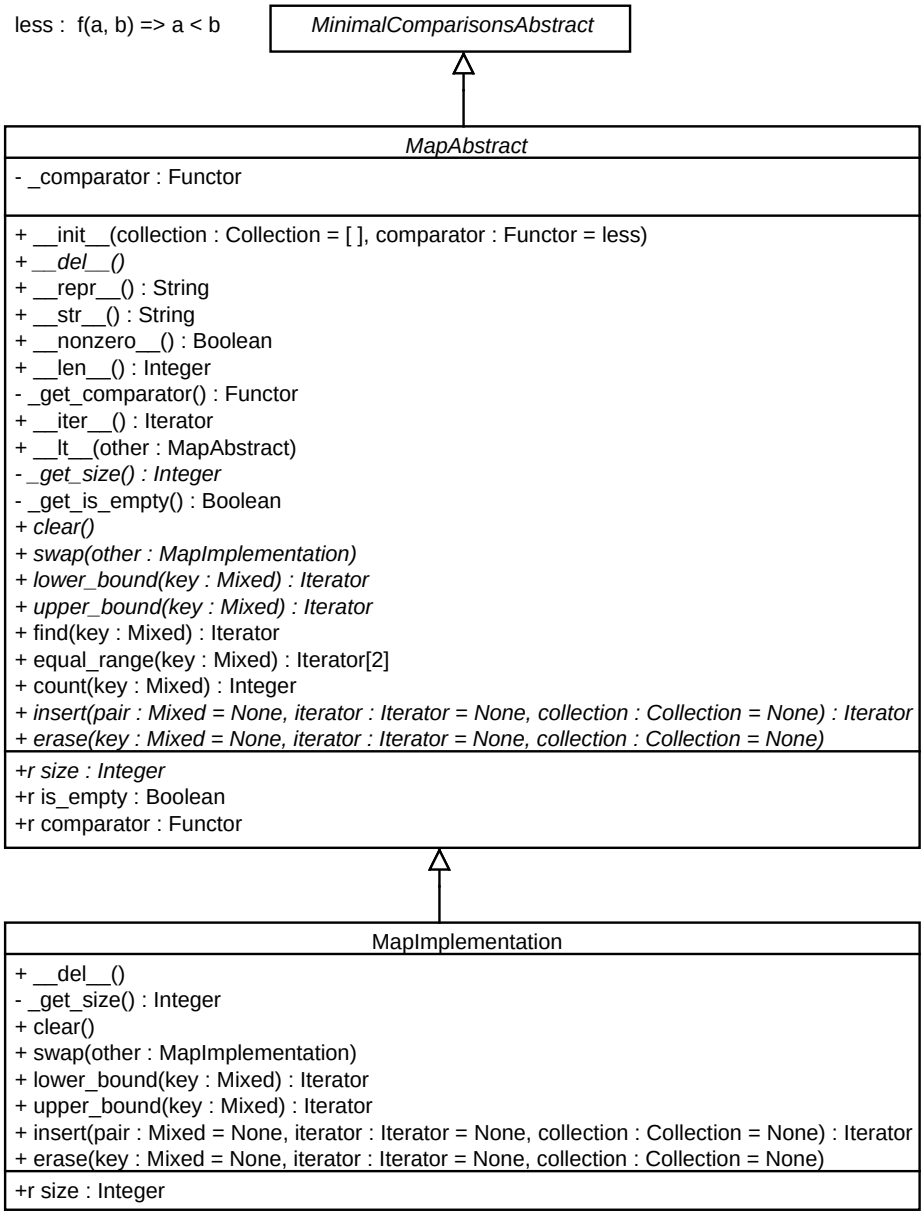
Figur D.6: Klassediagrammet for den abstrakte superklasse for sæt implementeringer.

D.7 Multisæt



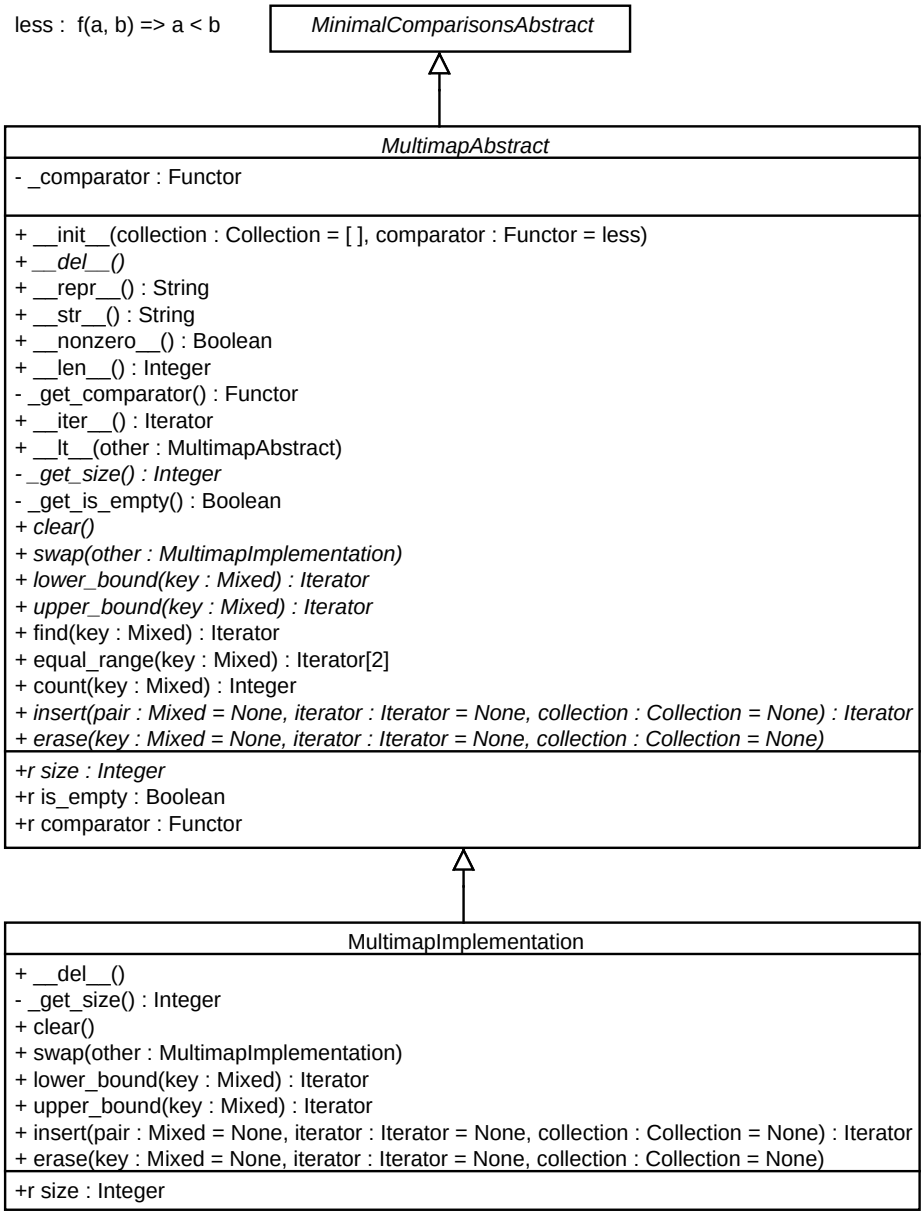
Figur D.7: Klassediagrammet for den abstrakte superklasse for multisæt implementeringer.

D.8 Kort



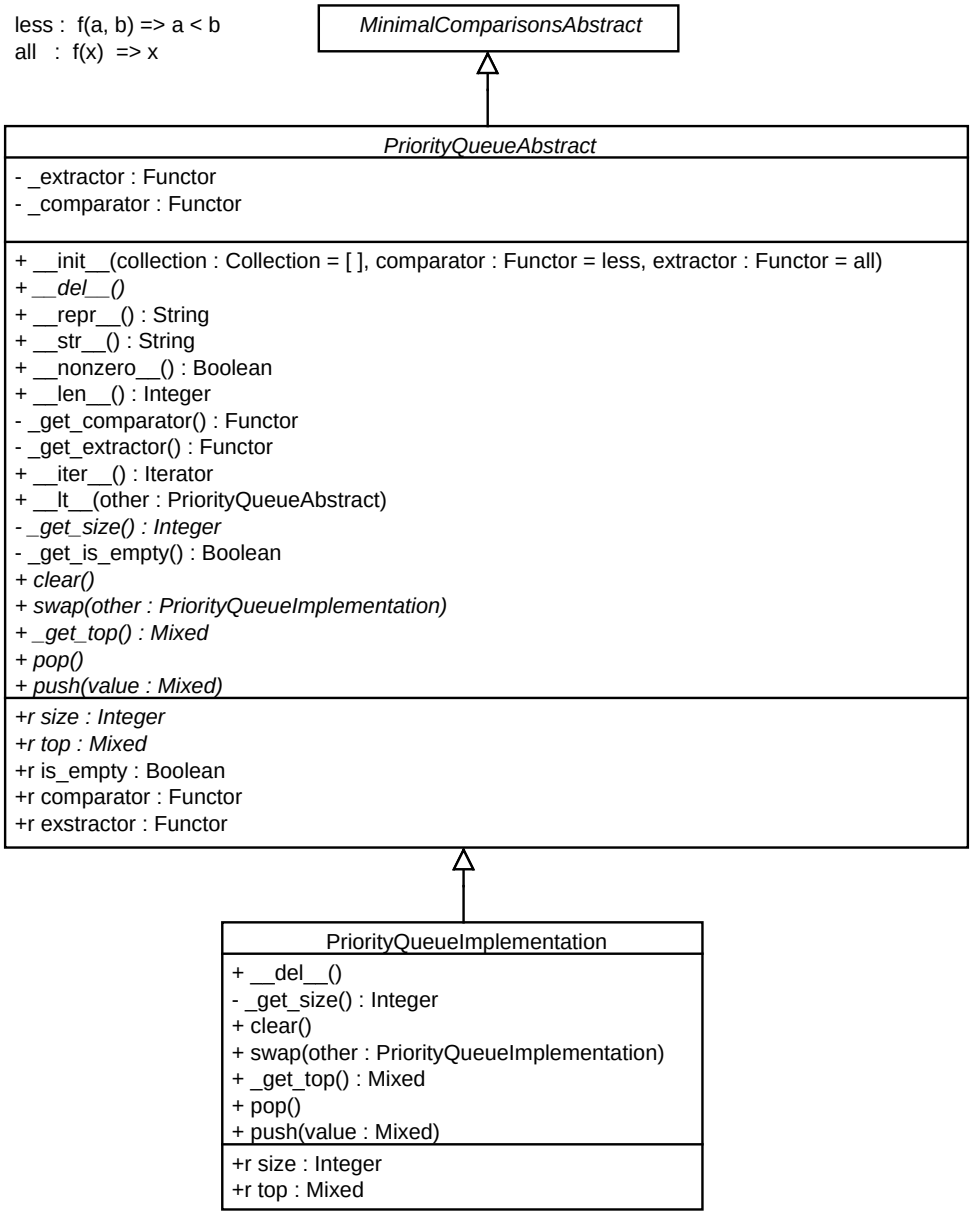
Figur D.8: Klassediagrammet for den abstrakte superklasse for kort implementeringer.

D.9 Multikort



Figur D.9: Klassediagrammet for den abstrakte superklasse for multikort implementeringer.

D.10 Prioritetskø



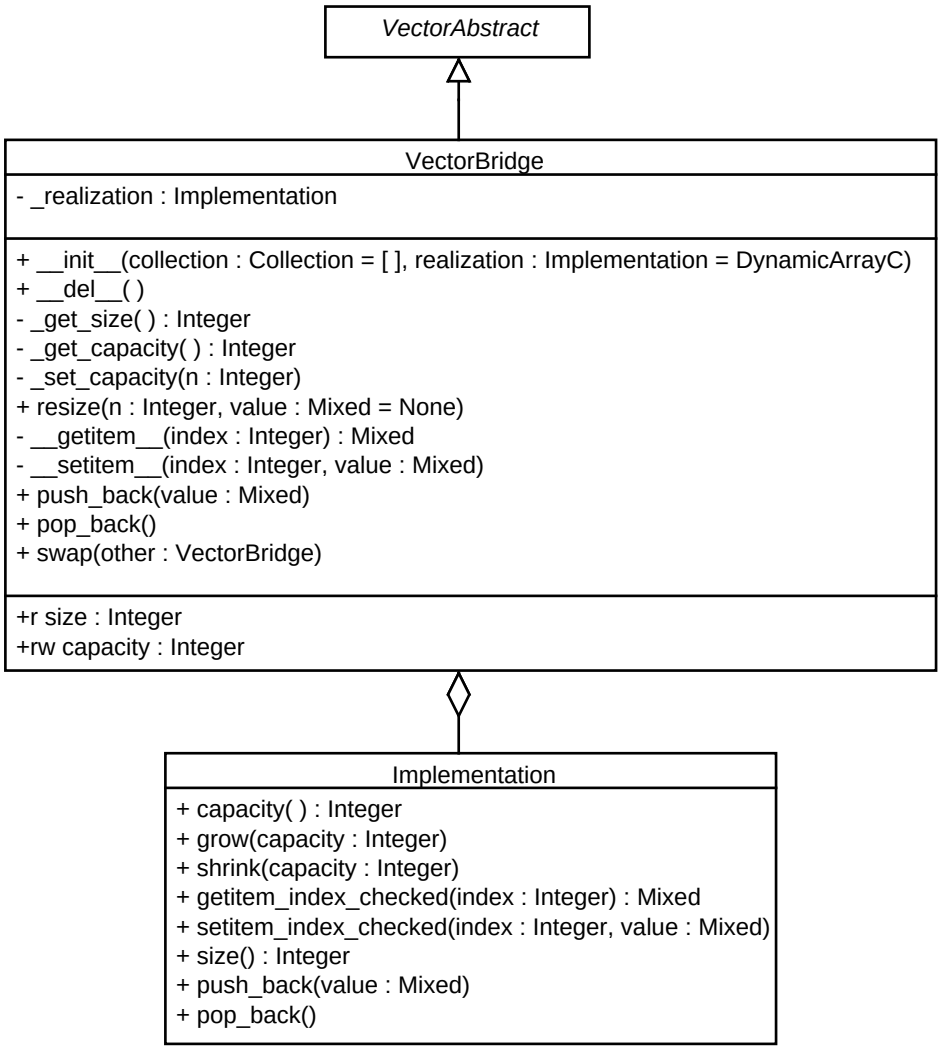
Figur D.10: Klassediagrammet for den abstrakte superklasse for prioritetskø implementeringer.

Bilag E

Broklasser for beholdertype

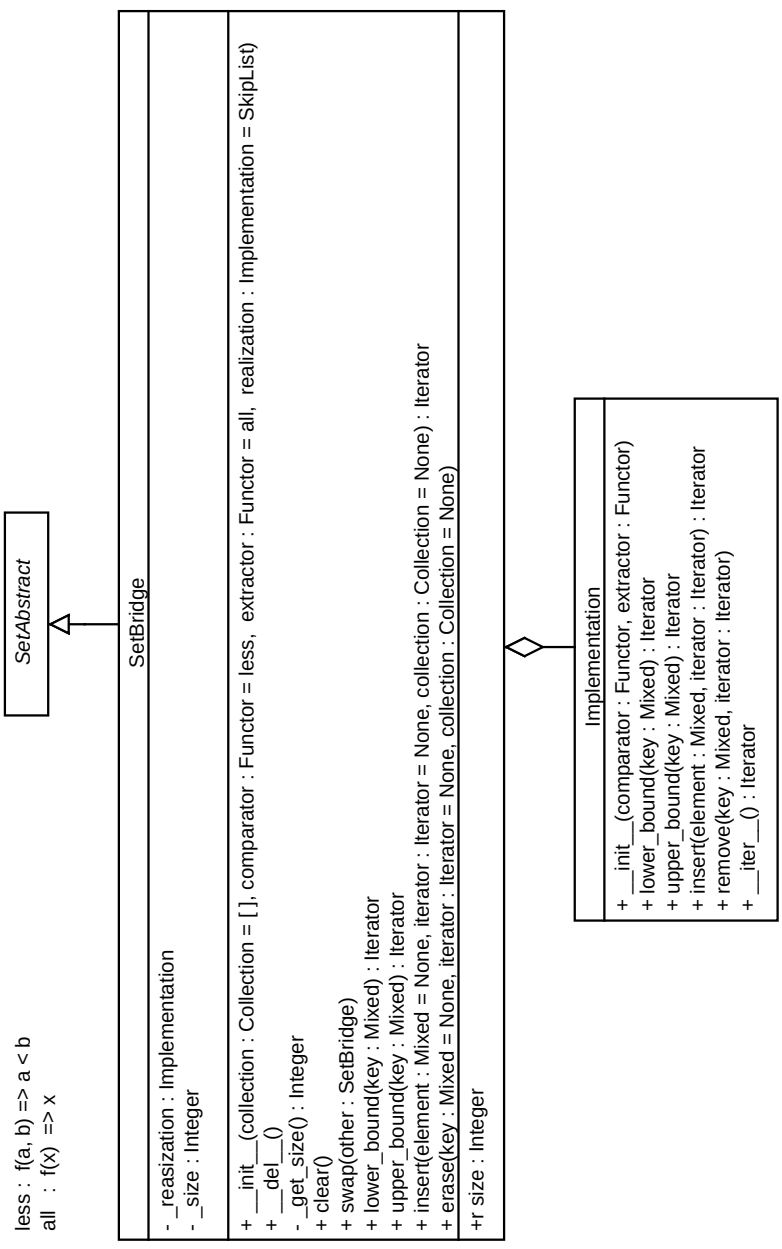
Dette bilag indeholder klassediagrammerne for de broerne til beholdertyperne vektor, sæt, kort, multisæt, multikort og prioritetskø i CPH collections. Det er kun de abstrakte- og broklasserne, der findes i CPH collections, mens klasserne vist som implementeringssklasser kun er der, for at vise hvilke metoder implementeringsklasse skal implementere.

E.1 Vektor



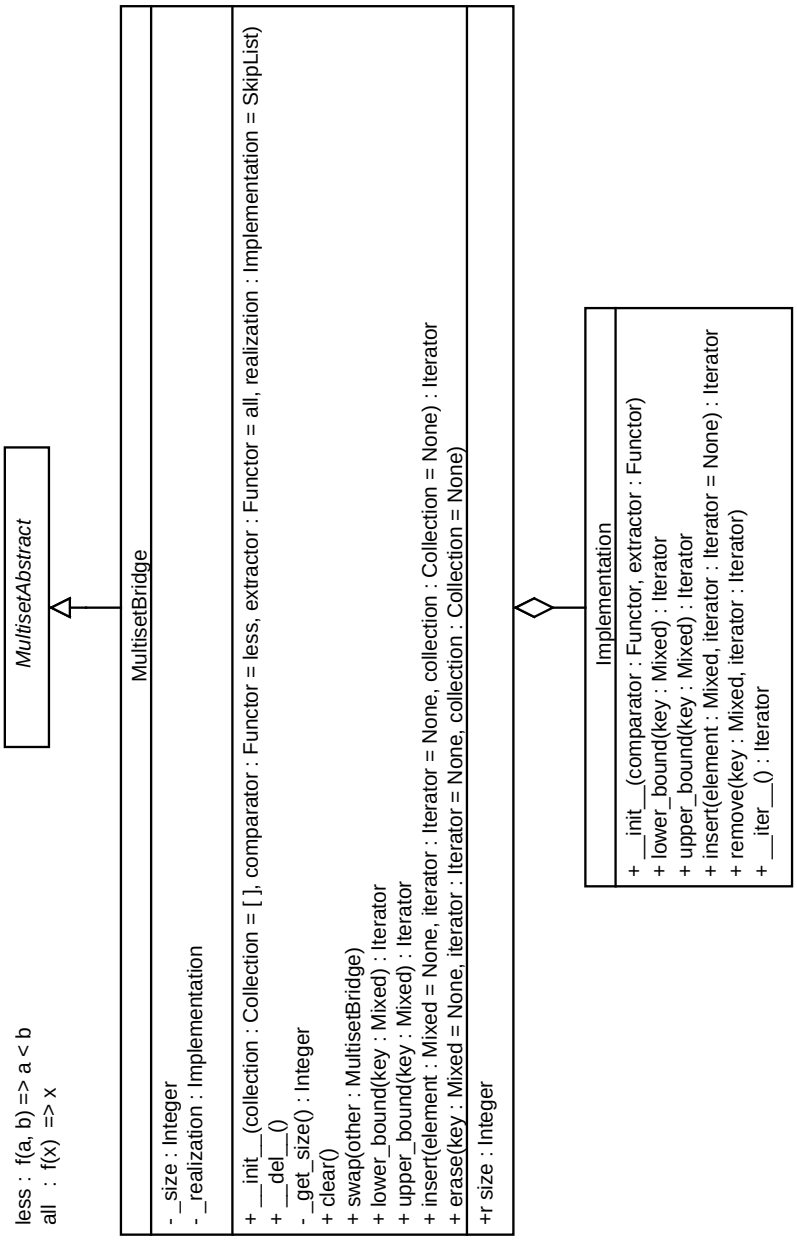
Figur E.1: Klassediagrammet for vektorbroen.

E.2 Sæt



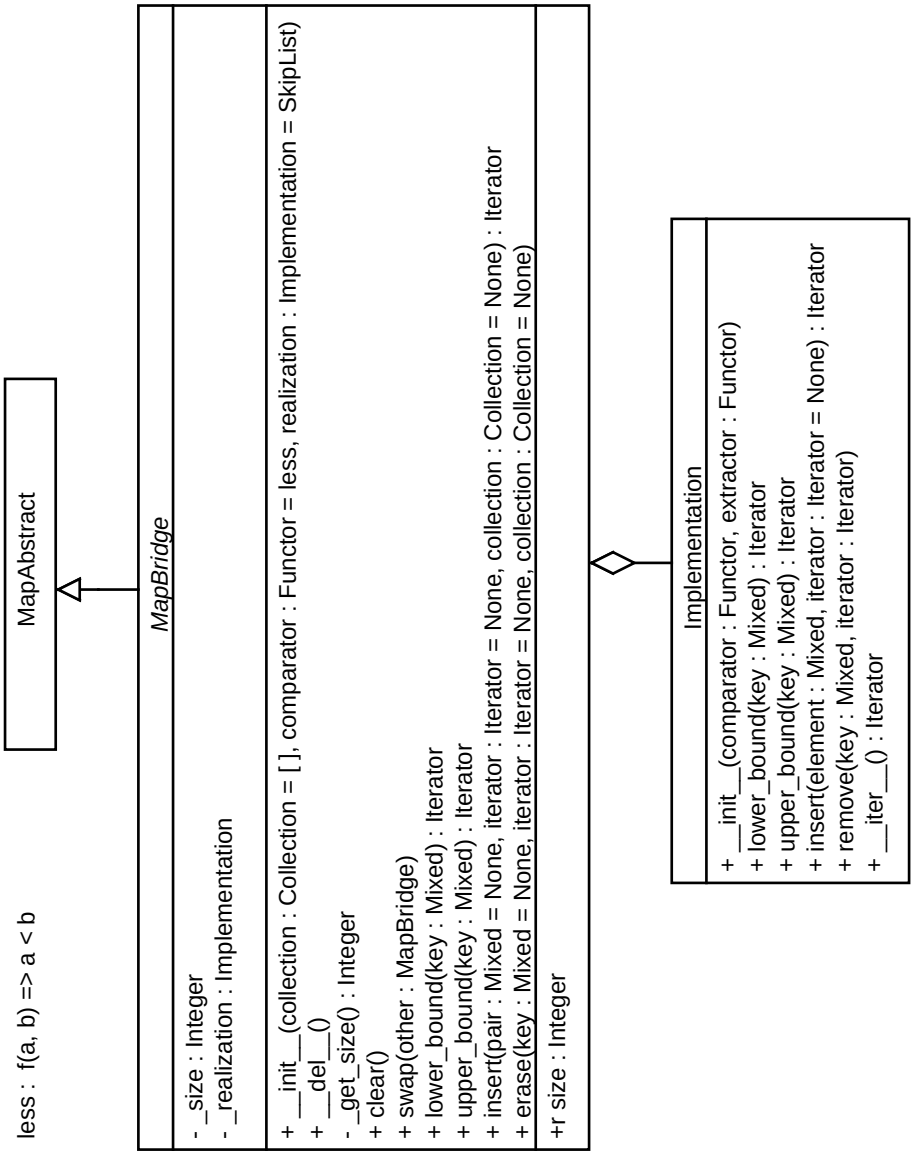
Figur E.2: Klassediagrammet for sætbroen.

E.3 Multisæt



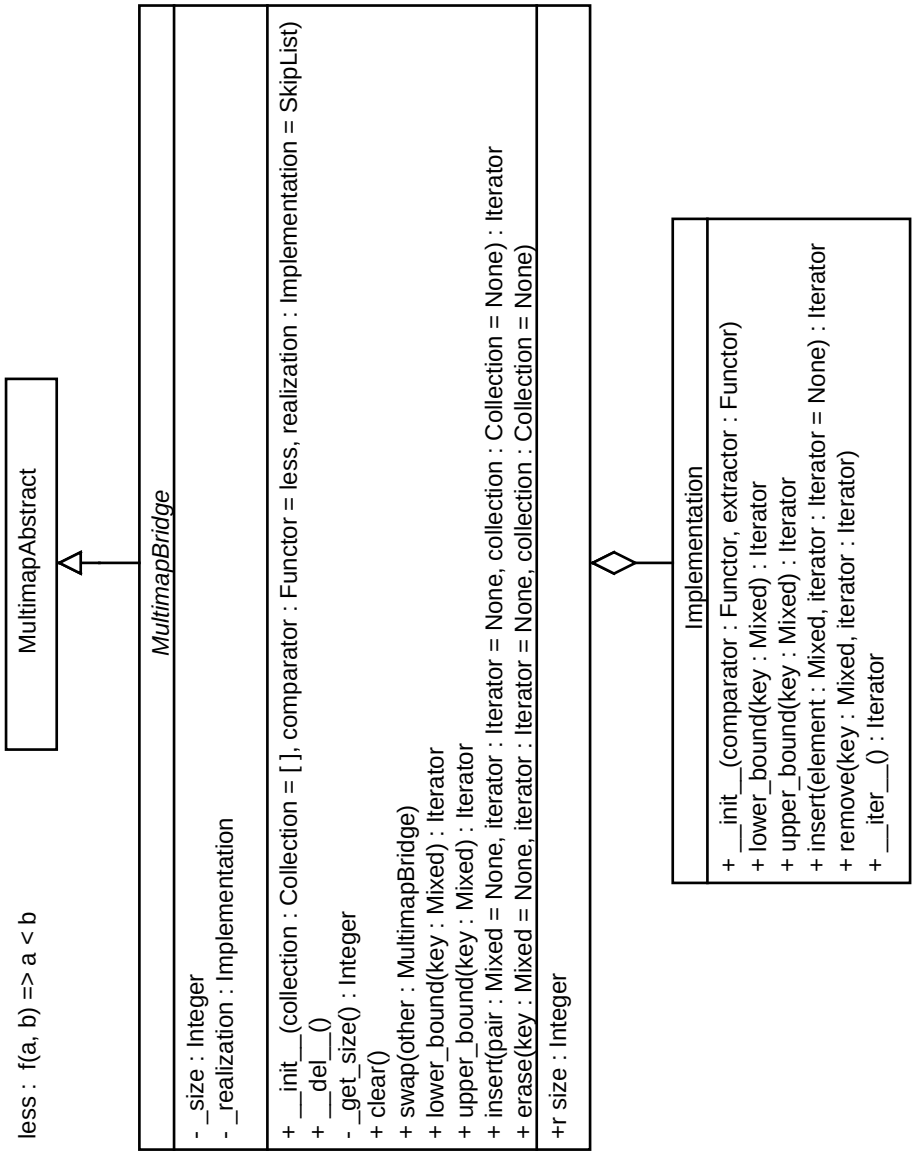
Figur E.3: Klassediagrammet for multisætbroen.

E.4 Kort



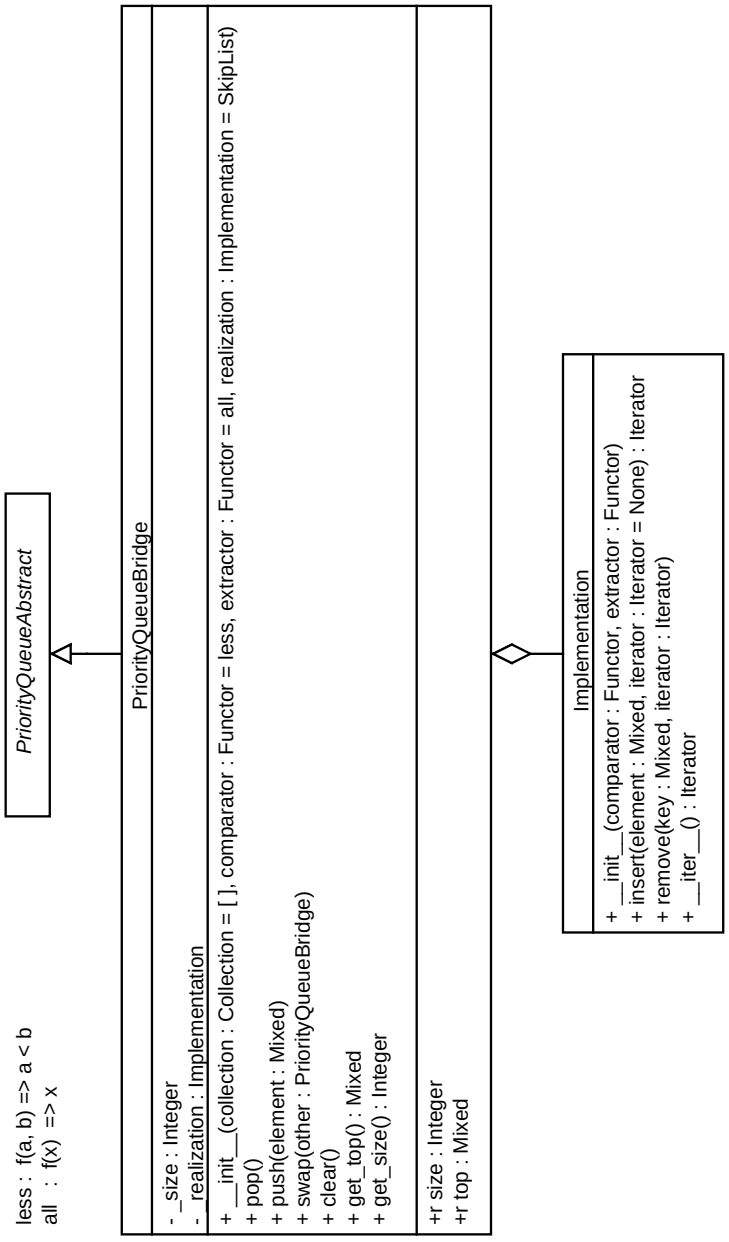
Figur E.4: Klassediagrammet for kortbroen.

E.5 Multikort



Figur E.5: Klassediagrammet for multikortbroen.

E.6 Prioritetskø



Figur E.6: Klassediagrammet for prioritetskøbroen.

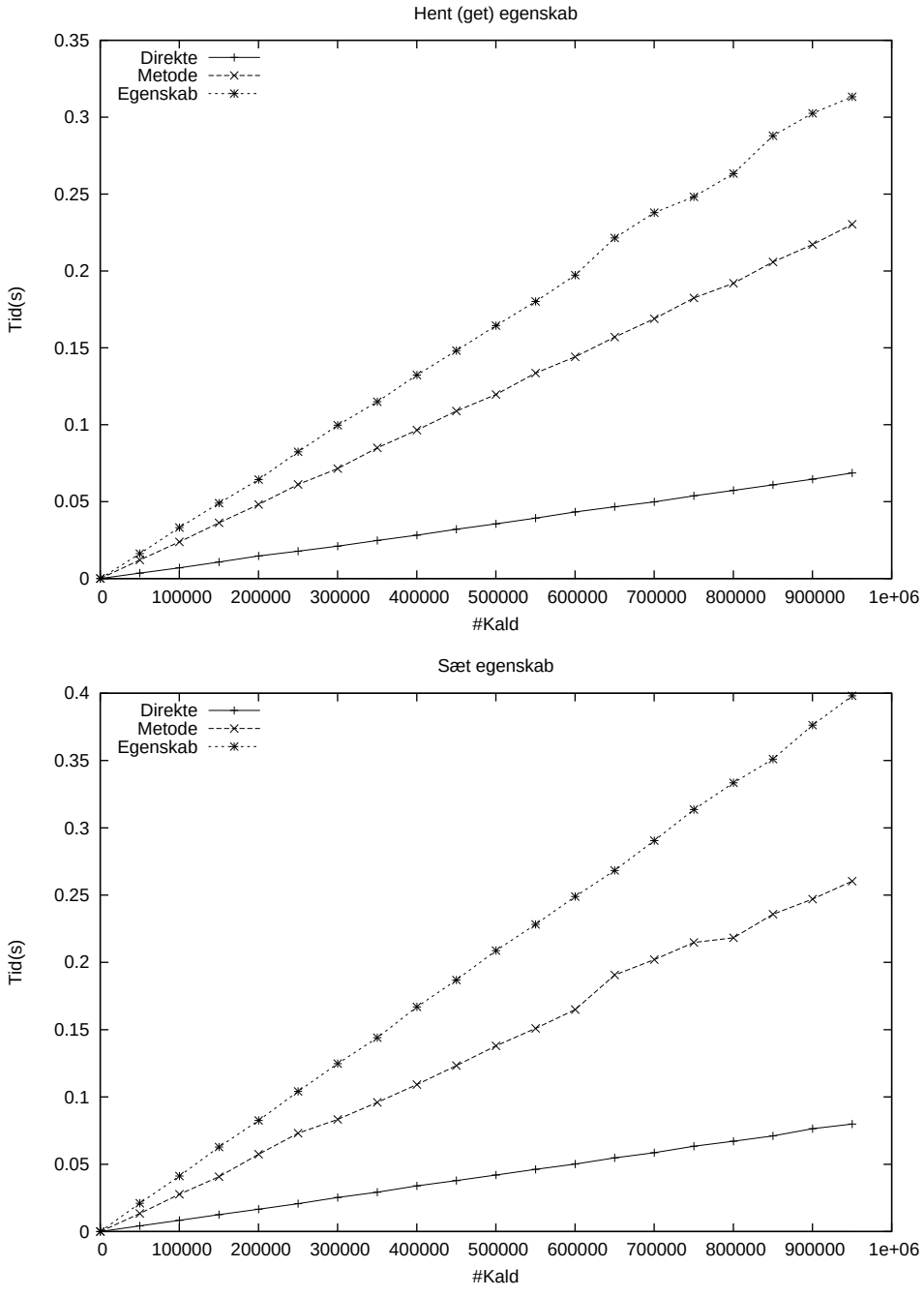
Bilag F

Ydelsestest

Dette bilag indeholder koderne og resultaterne af ydelsestest foretaget i specialet. Resultaterne af testene er grafer som *gnuplot* generer udfra tekstfilen (.gp). Tekstfilerne genereres af testprogrammerne. De oprindelig tekstfiler er tilgængelige i de vedlagte filer.

F.1 Intern tilgang via egenskaber

F.1.1 Resultat



Figur F.1: Tidsforbruget af forskellig teknikker til tilgang af data; direkte, metode og egenskab.

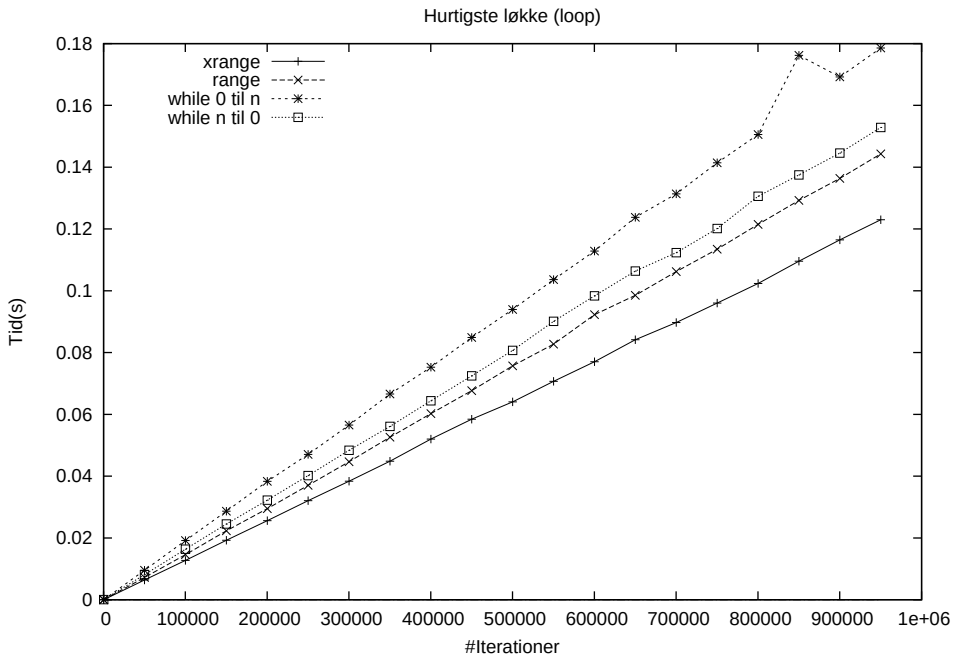
F.1.2 Kode

```
# -*- coding: cp1252 -*-
import copy

import cph_collections.container
import cph_collections.algorithm
from bench_prog import BenchmarkFunc


class HurtigsteLoop(BenchmarkFunc):
    def __init__(self, *args, **kwds):
        self._title = "Hurtigste løkke (loop)"
        self._xaxis = "#Iterationer"
        self._yaxis = "Tid(s)"
        self._filename = "bench_loop"
        super(self.__class__, self).__init__(*args, **kwds)

    def run(self, impl, i):
        return self.run_test(impl.run, i)

    def initialize(self, impl, i):
        return impl()

class Base:
    def verify(self, n):
        return self.c == n

class While0toN(Base):
    def run(self, n):
        i = 0
        p = 0
        self.c = 0
        while i < n:
            self.c += 1
            i += 1


class WhileNto0(Base):
    def run(self, n):
        i = n
        self.c = 0
        while i:
            self.c += 1
            i -= 1


class ForRange(Base):
    def run(self, n):
        self.c = 0
        for i in range(n):
            self.c += 1

class ForXrange(Base):
    def run(self, n):
        self.c = 0
        for i in xrange(n):
            self.c += 1

types = [
    ['xrange', ForXrange],
    ['range',ForRange],
    ['while 0 til n',While0toN],
    ['while n til 0',WhileNto0]
    ]

repeats = 10
```

```
66  HurtigsteLoop(types, start = 100, end = 1000000, jump = 50000,
        repeats=repeats)
```

F.2 Itereringsteknikker

F.2.1 Resultat



Figur F.2: Tidsforbruget ved forskellige itereringsteknikker.

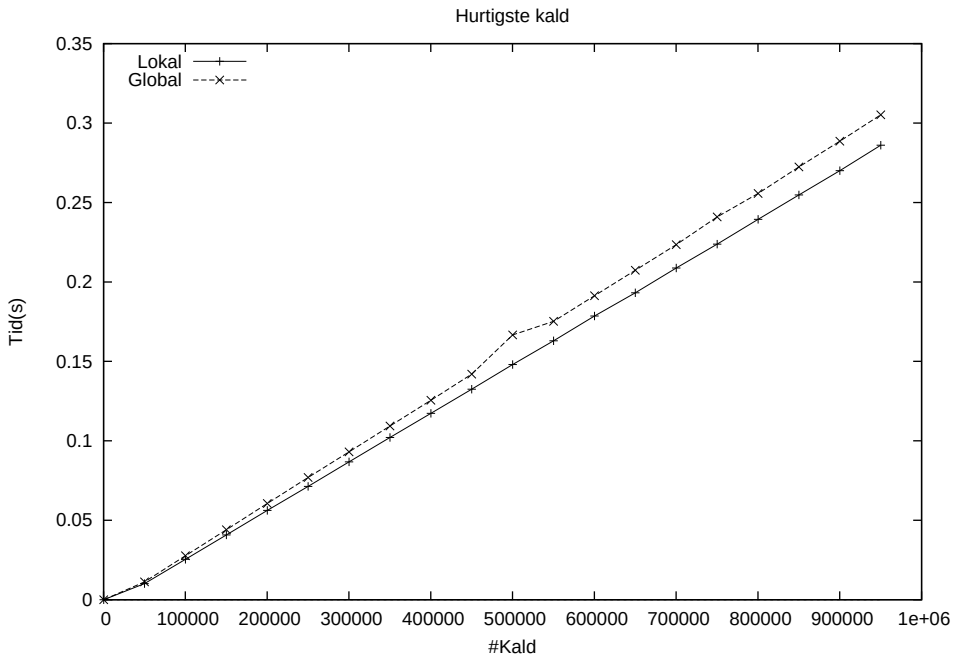
F.2.2 Kode

```
1  # -*- coding: cp1252 -*-
2  import copy
3
4  import cph_collections.container
5  import cph_collections.algorithm
6  from bench_prog import BenchmarkFunc
7
8
9  class HurtigsteLoop(BenchmarkFunc):
10     def __init__(self, *args, **kwds):
11         self._title = "Hurtigste løkke (loop)"
12         self._xaxis = "#Iterationer"
13         self._yaxis = "Tid(s)"
14         self._filename = "bench_loop"
15         super(self.__class__, self).__init__(*args, **kwds)
16
17     def run(self, impl, i):
18         return self.run_test(impl.run, i)
19
20     def initialize(self, impl, i):
```

```
21          return impl()
22
23  class Base:
24      def verify(self, n):
25          return self.c == n
26
27  class While0toN(Base):
28      def run(self, n):
29          i = 0
30          p = 0
31          self.c = 0
32          while i < n:
33              self.c += 1
34              i += 1
35
36
37  class WhileNto0(Base):
38      def run(self, n):
39          i = n
40          self.c = 0
41          while i:
42              self.c += 1
43              i -= 1
44
45
46  class ForRange(Base):
47      def run(self, n):
48          self.c = 0
49          for i in range(n):
50              self.c += 1
51
52  class ForXrange(Base):
53      def run(self, n):
54          self.c = 0
55          for i in xrange(n):
56              self.c += 1
57
58  types = [
59      ['xrange', ForXrange],
60      ['range',ForRange],
61      ['while␣0␣til␣n',While0toN],
62      ['while␣n␣til␣0',WhileNto0]
63      ]
64
65  repeats = 10
66  HurtigsteLoop(types, start = 100, end = 1000000, jump = 50000,
        repeats=repeats)
```

F.3 Lokale funktionkald

F.3.1 Resultat



Figur F.3: Tidsforbruget for funktions referencer.

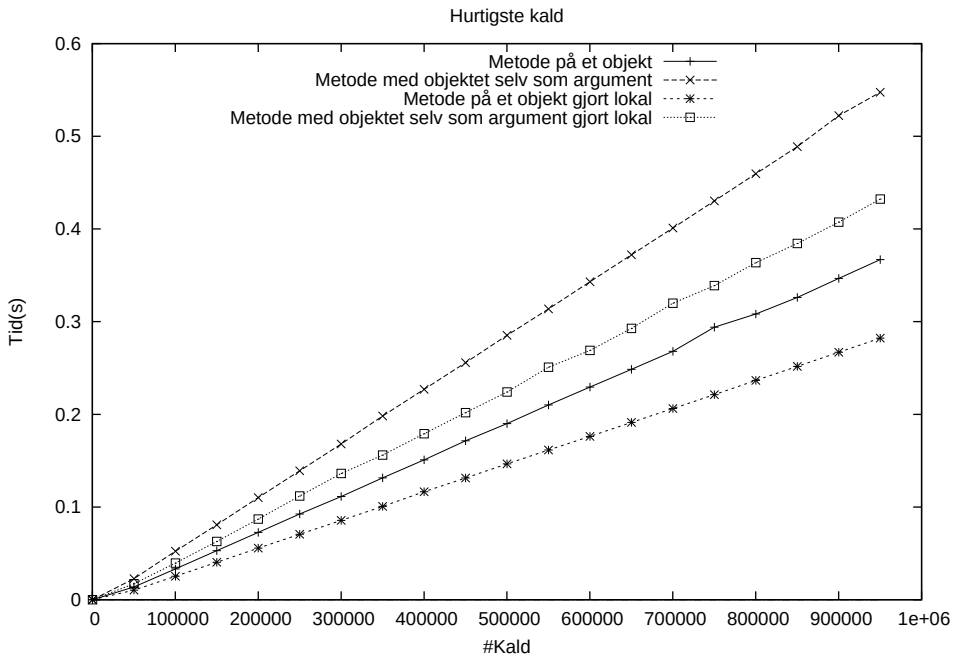
F.3.2 Kode

```
1  # -*- coding: cp1252 -*-
2  import copy
3
4  import cph_collections.container
5  import cph_collections.algorithm
6  from bench_prog import BenchmarkFunc
7
8
9  class HurtigsteKald(BenchmarkFunc):
10     def __init__(self, *args, **kwds):
11         self._title = "Hurtigste kald"
12         self._xaxis = "#Kald"
13         self._yaxis = "Tid(s)"
14         self._filename = "bench_function_local"
15         super(self.__class__, self).__init__(*args, **kwds)
16
17     def run(self, impl, i):
18         return self.run_test(impl.run, i)
19
20     def initialize(self, impl, i):
```

```
21          return impl()
22
23  def global_function(i):
24      return i * 2
25
26  class ForXrangeGlobal(object):
27      def run(self, n):
28          s = 0
29          for i in xrange(0, n):
30              s += global_function(i)
31
32  class ForXrangeLocal(object):
33      def run(self, n):
34          p = global_function
35          s = 0
36          for i in xrange(0, n):
37              s +=p(i)
38          return s
39
40
41
42
43  types = [
44      ['Local',ForXrangeLocal()],
45      ['Global', ForXrangeGlobal()]
46      ]
47
48  repeats = 10
49  HurtigsteKald(types, start = 100, end = 1000000, jump = 50000,
        repeats=repeats)
```

F.4 Lokale metodekald

F.4.1 Resultat



Figur F.4: Tidsforbruget for metode referencer.

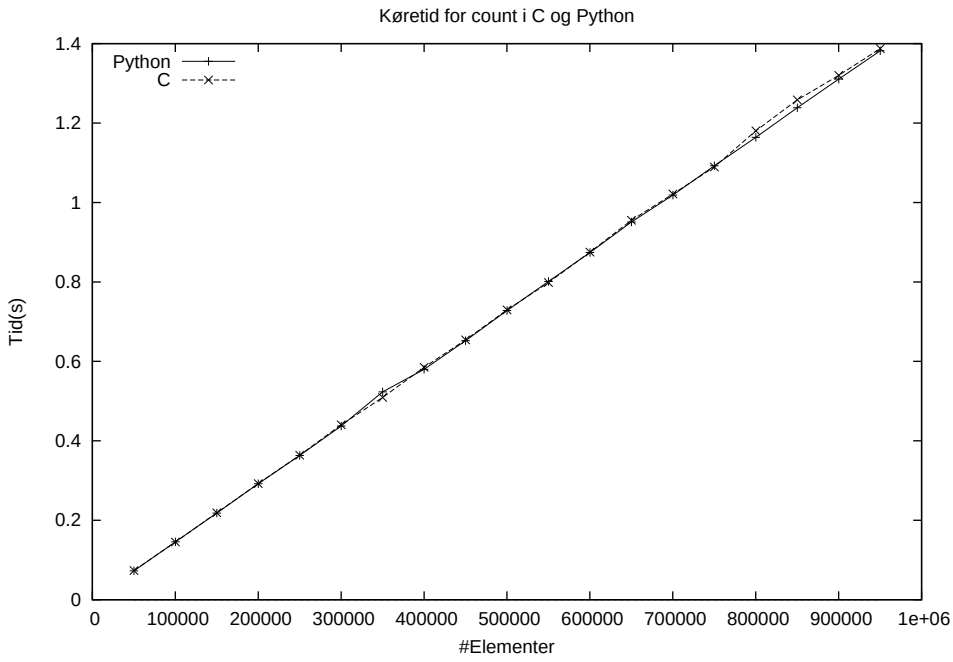
F.4.2 Kode

```
1  # -*- coding: cp1252 -*-
2  import copy
3
4  import cph_collections.container
5  import cph_collections.algorithm
6  from bench_prog import BenchmarkFunc
7
8
9  class HurtigsteKald(BenchmarkFunc):
10     def __init__(self, *args, **kwds):
11         self._title = "Hurtigste kald"
12         self._xaxis = "#Kald"
13         self._yaxis = "Tid(s)"
14         self._filename = "bench_method_local"
15         super(self.__class__, self).__init__(*args, **kwds)
16
17     def run(self, impl, i):
18         return self.run_test(impl.run, i)
19
20     def initialize(self, impl, i):
```

```
21          return impl()
22
23  class ForXrangeSelf(object):
24      def object_method(self, i):
25          return i * 2
26      def run(self, n):
27          s = 0
28          for i in xrange(0, n):
29              s += self.object_method(i)
30          return s
31
32  class ForXrangeSelfArgument(object):
33      def object_method(self, i):
34          return i * 2
35
36      def run(self, n):
37          s = 0
38          for i in xrange(0, n):
39              s += ForXrangeSelfArgument.object_method(self, i)
40          return s
41
42  class ForXrangeSelfLocal(object):
43      def object_method(self, i):
44          return i * 2
45
46      def run(self, n):
47          s = 0
48          p = self.object_method
49          for i in xrange(0, n):
50              s += p(i)
51          return s
52
53  class ForXrangeSelfArgumentLocal(object):
54      def object_method(self, i):
55          return i * 2
56
57      def run(self, n):
58          s = 0
59          p = ForXrangeSelfArgumentLocal.object_method
60          for i in xrange(0, n):
61              s += p(self, i)
62          return s
63
64
65
66  types = [
67      ['Method␣on␣self', ForXrangeSelf()],
68      ['Method␣with␣self␣as␣argument', ForXrangeSelfArgument()],
69      ['Method␣on␣self␣made␣local', ForXrangeSelfLocal()],
70      ['Method␣with␣self␣as␣argument␣made␣local', ForXrangeSelfArgumentLocal()]
71      ]
72
73  repeats = 10
74  HurtigsteKald(types, start = 100, end = 1000000, jump = 50000,
        repeats=repeats)
```

F.5 Køretid for `count` i `C` og Python

F.5.1 Resultat



Figur F.5: Tidsforbruget for `count` i `C` og Python.

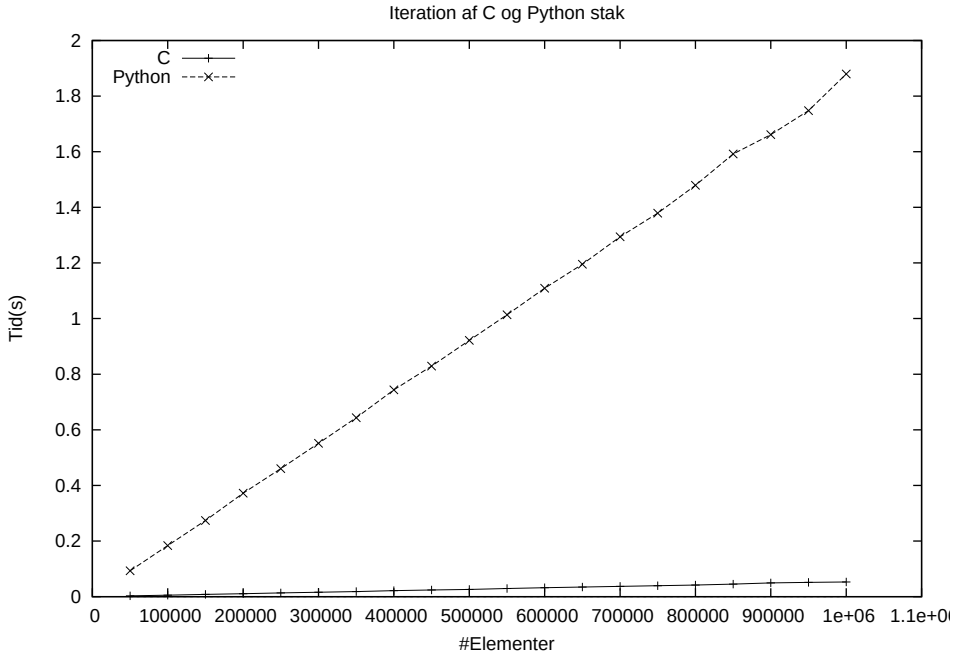
F.5.2 Kode

```
1  import time
2  import copy
3  from gnu_plot import GnuPlot, Plot
4
5  import cph_collections
6  import cph_collections.algorithm
7  from bench_prog import BenchmarkFunc
8
9
10 class CountBench(BenchmarkFunc):
11     def __init__(self, *args, **kwds):
12         self._title = "count␣#element"
13         self._filename = "bench_count"
14         super(self.__class__, self).__init__(*args, **kwds)
15
16     def run(self, impl, i):
17         return self.run_test(cph_collections.algorithm.count, impl, 0)
18
19 types = [
20     ['py:List', list().__class__]
```

```
21      ]
22
23  repeats = 3
24  CountBench(types, start = 50100, end = 1000000, jump = 50000,
        repeats=repeats)
```

F.6 Køretid for gennemløb af stak implementeret i C og Python

F.6.1 Resultat



Figur F.6: Tidsforbruget for gennenløb af stak implementeret i C og Python.

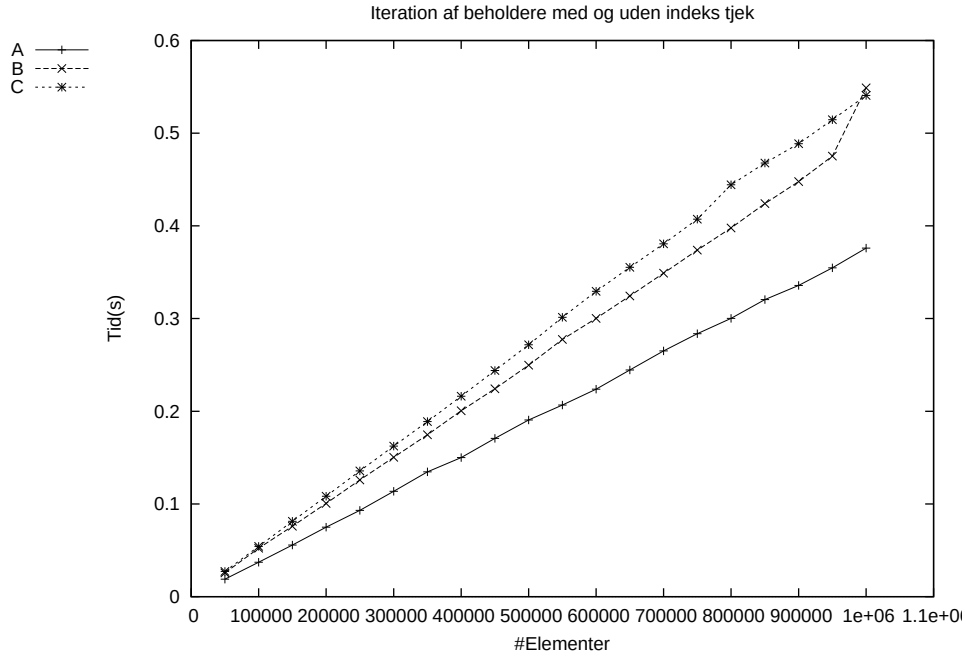
F.6.2 Kode

```
1  import time
2  import copy
3  from gnu_plot import GnuPlot, Plot
4
5  import cph_collections
6  import cph_collections.algorithm
7  from bench_prog import BenchmarkFunc
8
9  def count(impl):
10     p = 0
11     for e in impl:
12         p += e
13     return p
14
15 class Sum(BenchmarkFunc):
16     def __init__(self, *args, **kwds):
17         self._title = "Iteration af C og Python stak"
```

```
18          self._filename = "bench_stack"
19          super(self.__class__, self).__init__(*args, **kwds)
20
21      def initialize(self, impl_type, i):
22          return impl_type([1]*i)
23
24      def verify(self, got, i):
25          return i == got
26
27      def run(self, impl, i):
28          return self.run_test(count, impl)
29
30  types = [
31      ['C',cph_collections.Stack(impl='sc').__class__],
32      ['Python',cph_collections.Stack(impl='sl').__class__]
33      ]
34
35  repeats = 3
36  Sum(types, start = 50001, end = 1000001, jump = 50000, repeats=repeats)
```

F.7 Køretid for beholdere med og uden indeks test

F.7.1 Resultat



Figur F.7: Tidsforbruget for gennenløb af beholdere med og uden indeks tjek.

F.7.2 Kode

```
1  import time
2  import copy
3  from gnu_plot import GnuPlot, Plot
4
5  import cph_collections
6  import cph_collections.algorithm
7  from bench_prog import BenchmarkFunc
8
9  def test(impl):
10     l = len(impl)
11     p = 0
12     for i in xrange(l):
13         p += impl[i]
14     return p
15
16 class base(object):
17     def __init__(self, n):
```

```
        self._elements = [e for e in n]

    def __len__(self):
        return len(self._elements)

class A(base):
    def __getitem__(self, n):
        return self._elements[n]

class B(base):
    def __getitem__(self, n):
        if n < len(self._elements):
            return self._elements[n]
        raise Exception


class C(base):
    def __getitem__(self, n):
        if n > -1 and n < len(self._elements):
            return self._elements[n]
        raise Exception


class Index(BenchmarkFunc):
    def __init__(self, *args, **kwds):
        self._title = "Iteration af beholdere med og uden indeks tjek"
        self._filename = "bench_index"
        super(self.__class__, self).__init__(*args, **kwds)

    def initialize(self, impl_type, i):
        return impl_type([1]*i)

    def verify(self, got, i):
        return i == got

    def run(self, impl, i):
        return self.run_test(test, impl)



types = [
    ['A', A],
    ['B', B],
    ['C', C]
    ]

repeats = 3
Index(types, start = 50001, end = 1000001, jump = 50000, repeats=repeats)
```