Mini-project

Handout: problem formulation plus associated source code is now available via the CPH STL website (see CPH STL report **2005-1**)

Output: a progress report $\leq$ 12 pages (+ programs developed in the appendix)

Outcome: deeper understanding of basic standard library concepts (iterators, allocators, containers, etc.), deeper understanding of template programming, and an insight into library design (performance, exception safety, etc.)

Answer format: LaTeX `DIKU-article` style which is available via the CPH STL home page (the current version also supports Danish)

Handing in: electronically via ISIS by 23 June 2006 at 9.15

Meldable priority queue Q

$\mathcal{I}$ **find-max():** Return an iterator to the compartment which stores an element that, of all elements in Q, has the maximum value. The ordering criterion used is $\mathcal{C}$. If Q is empty, an iterator referring to the one-past-the-end element is returned.

$\mathcal{I}$ **insert($\mathcal{V}$ e):** Insert element e into Q and return an iterator to the compartment storing e for later use. If there is not space available to accomplish this operation, an iterator referring to the one-past-the-end element is returned.

void **delete($\mathcal{I}$ p):** Remove both the element stored at the compartment referred to by p and the compartment itself from Q.

void **increase($\mathcal{I}$ p, $\mathcal{V}$ e):** Replace the element stored at the compartment referred to by p with element e.

void **meld($meldable\_priority\_queue$ R):** Move all elements together with their compartments from R to Q thereby destroying R.

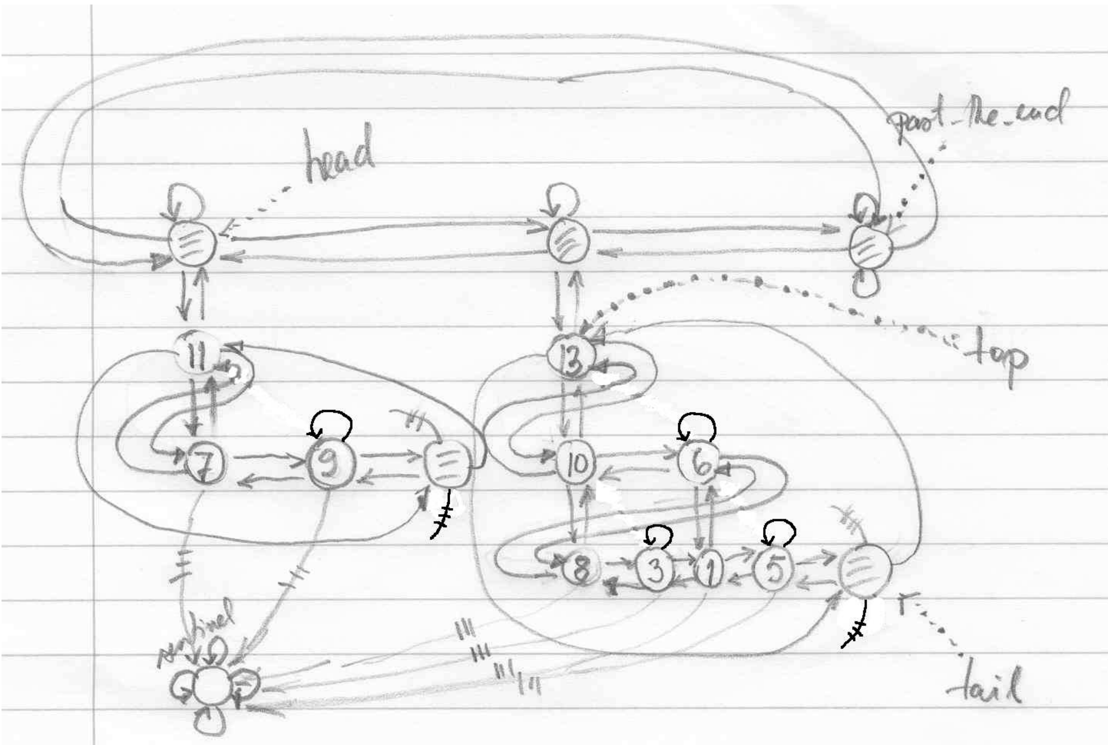
Challenges

- Fulfil the complexity requirements given in the C++ standard (if possible).
- Support **meld** at polylogarithmic cost.
- Support bidirectional iterators.
- Keep iterators valid at all times.
- Guarantee that iterator operations take $O(1)$ worst-case time.
- Avoid memory leaks.
- Guarantee exception safety.
- Be space-efficient.

Starting point

An almost complete implementation based on pointer-based binary heaps.

Example: A binary heap storing 10 integers.



Max-heapify (or `siftdown`)

```
template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::siftdown(node_type* r) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  node_type* p = left[r];
  while (!is_nil[p]) {
    node_type* q = right[r];
    if ((*this).comparator_(value[p], value[q])) {
      if ((*this).comparator_(value[r], value[q])) {
        (*this).swap_parent_right(r, q);
        p = left[r];
      }
      else break;
    }
    else {
      if ((*this).comparator_(value[r], value[p])) {
        (*this).swap_parent_left(r, p);
        p = left[r];
      }
      else break;
    }
  }
}
```

Alternatives to binary heaps

Data structure	Warnings
Binomial queues	
Brodal's heaps	very complicated
Fat heaps	
Leftist trees	
Maxiphobic heaps	
Multiary heaps	
Navigation piles	
Relaxed weak queues	complicated
Run-relaxed heaps	complicated
Space-efficient heaps	