Data structures and algorithms in a two-level memory

Ramzi Fadel　　Kim Vagn Jakobsen

Abstract

In many computer systems there is a big difference in the access time of the primary storage and the access time of the secondary storage. Since this difference does not seem to change in the future and the data sizes seem to increase, it is interesting to examine the behaviour of data structures and algorithms in a two-level memory. We describe a two-level memory model which is commonly used. We examine both the number of comparisons (*internal complexity*) and the number of block transfers between the two levels (*external complexity*).

Different solutions to the dictionary problem are analysed, among these a B-tree with both a good internal and external complexity. The priority queue problem is examined and we present a structure with a efficient external complexity. Sorting is examined and we present different algorithms with both optimal internal and external complexities. We examine two graph problems namely searching and the minimum-spanning-tree problem. Finally, we implement some sorting and priority queue solutions to establish the practicality of some of the algorithms.

Preface

This thesis has been made at the Department of Computer Science, University of Copenhagen, Denmark in the period from December 1995 to September 1996. The paper is a study of external-memory algorithms. We have selected problems which are relatively simple to define, but not necessarily have trivial solutions in external memory. Besides the algorithms specially designed for external memory, we have studied the behaviour of algorithms intended for an one-level memory model when used in a two-level memory. Finally, we have compared some of the algorithms using a two-level memory simulator.

We have received inspiration and help mainly from our thesis instructor, Jyrki Katajainen[1], who guided us in the right direction and made us think and write a lot. Besides we have received inspiration and suggestions from Jukka Teuhola[2] and Lars Arge[3]. Max Ottesen has carefully read the thesis and contributed with improvements to the way we write English. Thanks also to our friends and family for putting up with our nonsense.

Copenhagen, February 20, 1997
Ramzi Fadel
Kim Vagn Jakobsen

[1]Department of Computer Science, University of Copenhagen, Denmark
[2]Department of Computer Science, University of Turku, Finland
[3]Department of Computer Science, University of Aarhus, Denmark

Contents

Chapter 1

Introduction

A main reason for starting this project was an interest in a theoretical entry to practical aspects of algorithm execution and algorithm selection, and ensuing experimentation with different programs solving the same problem. That is answering questions like 'what algorithms can we expect to do well on existing computers and why?' and 'how can a theoretical lower bound be reached?'. Specially we will study algorithms in a hierarchical memory context. This means, that instead of just calculating the CPU-complexity, we also calculate the memory access complexity in separated memory.

1.1 Outline of the thesis

The plan of the project is to examine different algorithms for solving a specific problem, and to some extent examine implementations. This gives a natural separation of the thesis in a theoretical part and a practical part.

Chapter 2 *Two-level machine model* gives an introduction to different external-memory models and our own choice of model. Besides there are examples of analysing algorithms in the chosen model.

Chapter 3 and 4 *Dictionaries* and *Priority queues* respectively, are surveys of algorithms solving the two problems mentioned. An analysis is given in advance to forsee certain implementations as useful, and to select for practical evaluation later on. We develop a priority queue, which we denote I/O-optimal, since by using this data structure an I/O-optimal sorting algorithm is obtained.

In Chapter 5 *Sorting* we examine three different approaches to sorting: merging, partitioning and selection, and give several I/O-optimal algorithms. The results of Chapter 4 *Priority queues* are used when we are looking at selection sorting.

Chapter 6 *Graph problems* gives an overview of some results of external-memory graph algorithms. Many problems do not (yet) have optimal solutions which makes this area specially interesting.

Chapter 7 *Simulating external and virtual memory* describes the simulators we have developed.

Chapter 8 *Implementing sorting* Discourses eight sorting implementations and reprots the experimental results of their behavior graphically.

In Chapter 9 *Implementing priority queues* three implementations are presented and compared.

1.2 Notation

When presenting order of magnitudes we use the following notation for upper bounds, lower bounds and tight bounds:

Upper bound $O(f) = \{g \mid g(x) \leq cf(x) \text{ for all } x > x_0\}$

Lower bound $\Omega(f) = \{g \mid g(x) \geq cf(x) \text{ for all } x > x_0\}$

Tight bound $\theta(f) = \{g \mid \frac{1}{c}f(x) \leq g(x) \leq cf(x) \text{ for all } x > x_0\}$

where x_0 is given number.

When ever we use a logarithm function it denotes the expression: $\log_a x \equiv \log_a(\max(a, x))$.

Part I

Theoretical considerations

Before analysing algorithms we will examine how practice and theory are related and how we can choose a memory model which on the one hand reflects reality and on the other hand makes it possible to do theoretical calculations.

In the remainder of the part we have chosen different problems to examine. To each problem we have alternative solutions which are analysed within our memory model. The solutions are not merely chosen for the efficiency but also to show different types of approaches to the problem. Our primary goal is to analyse sorting and selected graph problems. To do this the problems of dictionaries and priority queues are dealt with first. At the end of each chapter we compare the different solutions and try to foresee there practicality.

Chapter 2

Two-level machine model

We will in this chapter justify our choice of a two-level memory model. First we briefly describe alternatives, and explain why we do not choose these models. Finally we analyse some simple data structures and algorithms within our model.

2.1 Real and modelled costs

In many theoretical analyses of algorithms movement of data is thought of as something simple. By simple we mean each assignment is counted as one time unit, without consideration to any movement of data within the memory of the computer, which can be a time consuming operation. The fact is that for even small amounts of data the storage is not linearly organized, but has some sort of hierarchical structure. A computer often has a constant number of registers which are the only directly accessible memory units. Other data are stored outside the registers and must be loaded into the registers, via the underlying levels, to be accessible (see Figure 2.1). The separated levels indicate that

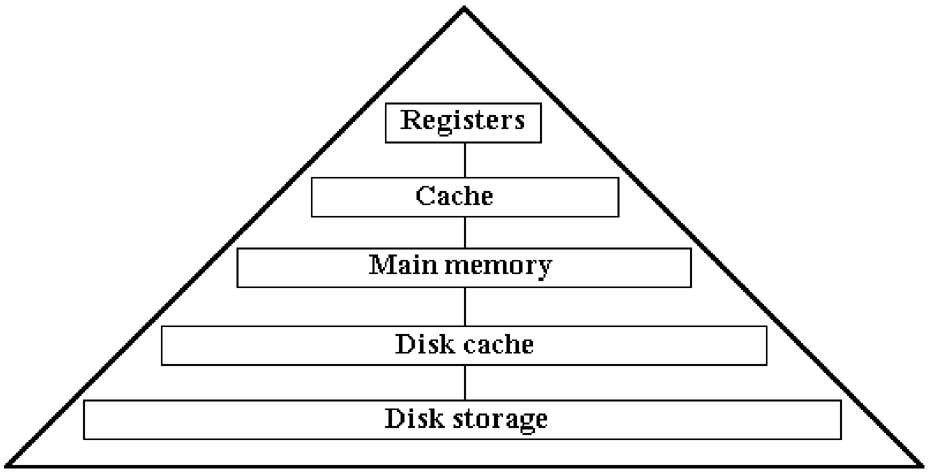

Figure 2.1: Example of hierarchical memory structure.

data currently placed on levels near the registers, are more easily loaded than data on the bottom level. Furthermore, data are usually not loaded one by one but are moved in blocks, so that the cost of moving a block of data is only a bit more expensive then that of moving a single data element. This is to exploit the fact that once an access to one part of data is acquired, the surrounding data are easily accessed immediately afterwards.

If the costs of block transports between the levels are small, there is no reason to care about these matters. But one block transfer is several times larger than one CPU operation, therefore we will consider these issues. Table 2.1 summarizes the behaviour of some typical storage media (figures from 1988), which could describe a 'real' computer system [42]. Making a complete evaluation when analysing algorithms that is taking all

Media	**Access time**	**Transfer block size**	**Capacity**
Registers	2 - 20 ns	4 bytes	$10 - 100$ bytes
Cache	20 - 200 ns	4 - 32 bytes	$100 - 10^5$ bytes
Main memory	200 - 600 ns	500 - 1000 bytes	$10^6 - 10^8$ bytes
Disk cache	2 - 4 ms	1000 bytes	$10^5 - 10^6$ bytes
Disk storage	10 - 100 ms	-	$10^8 - 10^{10}$ bytes

Table 2.1: Typical storage media features.

aspects into consideration, would be impossible. A real evaluation must be to measure the behaviour of a program implementing the corresponding algorithm. But instead of trying to make a complex analysis one can define a simplifying model which can give some information. The algorithms are then analysed within the model. The costs usually studied are the time complexity, meaning how long one has to wait before the work is done. This is not the same as the total work, since some of the work can be done in parallel.

The two following models use a cost function $f(x)$, which denotes the cost of accessing memory element x. It is justified that $f(x) = \lceil \log(x) \rceil$ is a natural choice, because it is discrete and pyramide-like, but other functions have been considered. The HMM_f (Hierarchical Memory Model) and BT_f (Block Transfer) models described in [2] and [3] respectively, are examples of simplifying models. Memory accesses are counted with the same units as other CPU-operations, but the costs of these accesses are not necessarily constants. The HMM_f uses $f(x)$ as a cost function. This model with an increasing function reflects the difference in the retrieval cost between near and far locations. The BT_f model is a generalization of the HMM_f model. The latter model is extended with the possibility of block transferring, so that moving n elements from location x costs $f(x)+n$ units. There are no boundaries to n except that a block can not cross level boundaries. The model assumes that once a location is accessed the surrounding locations are easily available. It has been shown that problems can be solved more efficiently in the BT_f model than in the HMM_f model.

Even though these models are somehow realistic we prefer a separated cost counting. The two things to be counted are the complexity of an algorithm (register operations) and accesses crossing the hierarchical borders. We assume, as most computer architectures are constructed today, that the transfers are carried out in blocks of fixed sizes. We will not specially analyse the space requirements of the algorithms because we assume that the external memory is sufficiently large. Besides, algorithms with extremely bad space complexities usually have bad time complexities.

2.2 The model

We are interested in a model that can be used practically when analysing algorithms, and at the same time reflect the difficulties of having separated memories. For these requirements one can consider a two-level system. The two levels are named the *internal* and *external* memory or primary and secondary memory respectively. The internal memory is virtually the first levels of Figure 2.1 including the registers, cache and main memory. The external memory covers the remaining levels. Both internal and external memories are divided into frames of equal size. Transfer of records between internal and external memories can only be done as whole blocks moving between these frames.

The central processing unit (CPU) can only directly access the internal memory. Therefore we have two kinds if costs. *Internal complexity* is defined as the number of reads and writes the CPU performs on the internal memory. This includes assignments, comparison and so on. This is the cost, usually denoted an algorithms complexity. Accessing data that is not in the internal memory requires a transfer of a block to the internal memory. A *block read* copies a block of data from a frame in the external memory to a frame in the internal memory. A *block write* copies a block from a frame in internal memory to a frame in external memory. We define the *external complexity* (or *I/O-complexity*) as the number of block reads and writes. For example, the I/O-complexity for scanning a file systematically from beginning to end with one disk and one processor is proportional to the number of blocks occupied by the file (only block reads are performed). While the internal complexity is still proportional to the size of file.

The following symbols describe the two-level model [44]

B Block size, the number of records transported simultaneously between internal and external memory. We call the moved entity a B-*block* or just a *block* or *page*.

N, n Problem size and the number of blocks required respectively. The problem 'size' can denote, for example, the size of input, the current size of a data structure or the total number of operations performed.

M, m Internal memory size in records and in blocks respectively. An m-block is a collection of m B-blocks that after the loading can be processed in internal memory without extra I/O costs.

P Number of disks.

P' Number of processors.

We always assume that $1 < B < M < N$. In many analyses it is required that $n > 2m$. Parallelism is present in three ways. Simultaneous read/write of B records, the possibility of P simultaneous block transfers and finally parallel processing. The I/O-complexity of scanning can now be expressed as $\theta(N/BP) = \theta(n/P)$, and the internal processing time is $\theta(N/P')$. We shall restrict ourselves to the case where $P = 1$ and $P' = 1$, meaning that the I/O-complexities are proportional to the total external work, and the internal complexity is proportional to the total internal work.

2.3 I/O-complexity

In the rest of the paper we use the term *I/O-complexity* meaning the magnitude of block transfers between the to levels of memory, and *I/O* meaning one block transfer. The magnitude of the I/O-complexity is determined both by the algorithm processing the data, and by the administration of the blocks in and out of the internal memory. We will use the terms *internal algorithm*, or *virtual algorithm*, about algorithms that assume some sort of memory management, that is, a virtual memory. The problem of using virtual memory is that a page replacement strategy is not designed specially for a specific algorithm, which can cause the replacement of a block needed soon again, instead of replacing a block not needed anymore. *External algorithms*, on the other hand, handle all the block transfers themselves as well as the data processing. This strategy should give the best administration of the block transfers between the two levels of memory, but programming can be quite complicated. Therefore one often relies on the operating system to support paging when designing an algorithm. This means that the programmer, to begin with, does not have to worry about the memory management. In the best case, this could result in an algorithm which is as efficient as a corresponding external algorithm. There is thus a trade-off between the runtime complexity and the programming complexity. This does not apply to simple problems such as scanning, but we will show in the following chapters how there are many ways to solve problems giving different I/O and programming complexities. How much one can gain by choosing external algorithms compared to internal algorithms with virtual memory cannot always be settled a priori.

As an example of different algorithms, concerning the same problem, leading to different I/O-complexities, consider the touching problem where the task is to touch each record. Even though the lower bound on the I/O-complexity for touching is $\Omega(n)$, one can easily design an algorithm with the optimal internal complexity, $\theta(N)$, but with the non-optimal I/O-complexity $O(N)$. The difference is in the order of accesses, determined by the two algorithms. In the optimal algorithm the data are scanned from beginning to end so that each block is accessed only once. The non-optimal algorithm touches the records in an arbitrary order but each record only once, meaning that each block can be accessed B times. Since there are about N/B blocks this gives an external complexity of $O(N)$.

The problem of permuting N records has a lower bound internal complexity which is linear $\theta(N)$. The lower bound I/O-complexity has been shown to be $\Omega(\min(N, n \log_m n))$ in [4] which in practice means $\Omega(n \log_m n)$. Therefore internal and external complexities are not always proportional.

As another example of the relation between internal and external complexities, consider m-way merging of N records. This operation has an I/O-complexity of $\theta(n)$, but a non-linear internal complexity of $\theta(N \log_2 m)$ (see Section 2.4.4).

It is worth noting that the smallest complexity when speaking of I/O-complexities is not $O(1)$. This figure would mean a constant number of I/O's per problem instance. The smallest complexity is rather $O(\frac{1}{B})$ which means a constant number of I/O's per $O(B)$ problem instances. This fraction should be understood as an average over a series of operations, as for example in touching where the total number of I/O's is $\theta(n)$ and therefore the number of I/O's per input element is $\theta(\frac{n}{N}) = \theta(\frac{1}{B})$. The I/O-complexity $O(1)$ is no longer a constant since $O(1) = O(B\frac{1}{B})$. In general we say that a complexity is *amortized* if the complexity can only be achieved by performing a series of operations.

2.4 Programming in two-level memory

In this section we give examples of problems and external algorithms that solve the problems. It is intended as an introduction to programming and analysing within the two-level memory model.

2.4.1 Stacking

We will here show an implementation of a *stack* which supports the operations :

Push(x,S) Inserts record x as new top record in stack S.

Pop(S) Removes the top record of stack S

Buffering is an useful way to achieve an external complexity of $O(\frac{1}{B})$ per operation, while the internal complexity is $\theta(1)$. When performing N operations on a stack the external complexity can also be expressed as $O(\frac{N}{B}) = O(n)$. A buffer size of two blocks of internal memory are needed to reach this bound. Using more blocks (for example m blocks) does not improve the worst case bound.

The internal memory will have a buffer containing the top records (the top blocks). The external memory will contain the remaining blocks of data, in such a way that the next block can be found easily. Figure 2.2 illustrates a buffered stack. The thin lined box is the internal buffer of two blocks. The gray areas are stack records, while the white areas are empty spaces. First, we describe why a buffer of one block is not enough to

Figure 2.2: Stack

reach the wanted external complexity. When having this buffer size, every time the buffer either becomes full or empty a block transfer occurs. One could then be evil and cause a block transfer on every second push and pop, giving an external complexity of $O(1)$ per operation.

We now show the implementation with a buffer of two blocks. A block read occurs when a pop is executed on an empty buffer, see Figure 2.3. Once the wanted block is read, there will be at least B operations before one more block read can occur. Notice that two

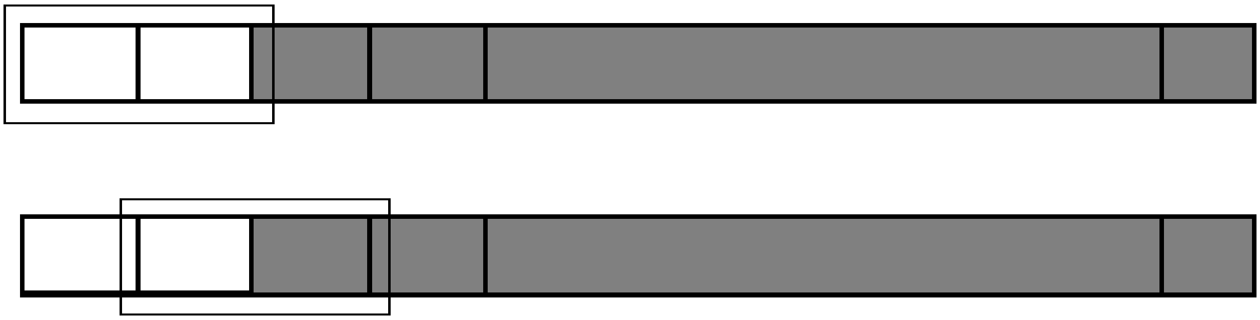

Figure 2.3: Before and after a pop operation on an empty buffer

pushes will not cause the newly read block to be written back. What will happen is that the empty block (already in internal memory) will be used without writing the old top block to external memory. Analogously, a block write happens when a record is pushed on a full buffer, see Figure 2.4. There will at least B operations until another block write happens. The internal complexity is the usual $\theta(1)$ for each operation. Since at most every Bth operation will cause one I/O, one operation will have an amortized external complexity of $O(\frac{1}{B})$. If we have a buffer of m blocks ($= M$ records) the m top blocks can be kept in internal memory. This buffer size gives an external complexity of $O(n - m)$, since at least m blocks of records are never written to external memory. If $n > 2m$ the

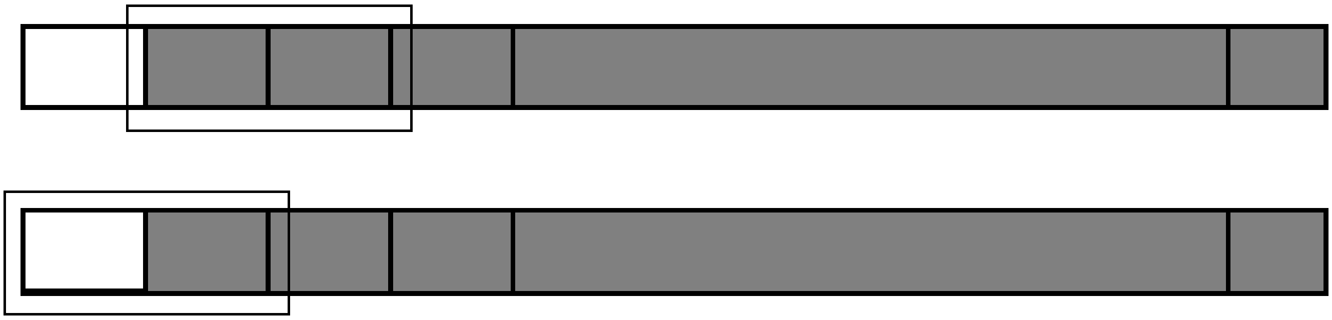

Figure 2.4: Before and after a push operation on a filled buffer

external complexity becomes $O(n)$. In practice it might be useful to keep more than two blocks in internal memory. For example, if during the execution we often perform the pop operation K times in a row, the buffer should be of size $\lceil \frac{K}{B} \rceil$.

2.4.2 Queuing

A *queue* data structure supports the following two operations:

In-queue(x,Q) Inserts a record x as last record of queue Q.

Out-queue(Q) Removes the first record from queue Q.

The structure requires at least two blocks of buffering to achieve the the external complexity of $O(\frac{1}{B})$ per operation. One buffer for insertions and another for the records to be removed. When having only two blocks of buffer in internal memory every Bth insertion causes a block write to external memory, and every Bth removal causes a block read. We can conclude that one operation has an external complexity of $O(\frac{1}{B})$.

If one uses a bigger buffer, of, for example, m blocks, the external complexity is $O(n-m) = O(n)$ for N operations and $n > 2m$. This means that only two buffer blocks are needed.

2.4.3 Two-way merging

The task of merging two sequences requires three internal blocks. Two for each input sequence and one as an output buffer. When the number of records to be merged is N, the internal complexity is $\theta(N)$. The input sequences are assumed to be placed continuously in at most $n+2$ blocks. The number of block reads is therefore $\theta(n)$. The number of writes is also $\theta(n)$, since the size of the output for N records is $\lceil \frac{N}{B} \rceil = \theta(n)$. The I/O-complexity is thus $\theta(n+n) = \theta(n)$.

2.4.4 Multi-way merging

In the implementation of stacking and queuing, more buffer space in internal memory did not improve the theoretical bounds of the external complexities, whereas in multi way merging a larger buffer improves the worst case bounds.

The problem is to merge k sequences containing N records in total. If we use two-way merging, as described above, $\log_2 k$ times, the internal complexity will be $\theta(N \log_2 k)$ and the external complexity will be $\theta(n \log_2 k)$. But if we use $\theta(m)$ input buffers we can merge $\theta(m)$ sequences at a time, giving an external complexity of $\theta(n \log_m k)$. Especially if we perform a m-way merge, that is $k = \theta(m)$, the external complexity will be $\theta(n)$. The internal complexity can be kept as $\theta(N \log_2 k)$ if one builds a logarithmic selection structure. Deciding from which sequence the next element should be taken, requires $\theta(\log_2 m)$ time, hence the internal complexity is $\theta(N \log_2 m \log_m k) = \theta(N \log_2 m \frac{\log_2 k}{\log_2 m}) = \theta(N \log_2 k)$. The selection structure is built by using $\theta(m)$ pointers to the input lists.

2.5 Related work

As mentioned in Section 2.1 there exist different memory models. Apart from the HMM_f and the BT_f models ([2, 3]), we have seen the following descriptions.

In [45] the $P - HMM$ and $P - BT$ models presented. These are combinations of the HMM_f and BT_f models respectively and the PRAM model. That is, a PRAM model where the local memories are either HMM_f or BT_f memories.

The $P - HMBT$ and $PP - HMBT$ models are defined in [29], and are also based on the BT_f model. The $P - HMBT$ extends the BT_f model by allowing parallel block transfers, i.e. each level can transfer one block simultaneously. The $PP - HMBT$ model further more allows pipelining between the levels.

In [36] a model is designed which is based on the physical characteristics of RAM storage, $LT - RAM$ (Limiting-Technology). The memory cells are placed in an array, and the cost of referring to a single cell is proportional to the number of rows of the array. It is shown that algorithms with local characteristics work better in this model than the 'obvious' ones.

We have in our model assumed that each I/O has a cost of one unit, which allows us to compare algorithms analytically. In [20] three charging schemes are used:

Full-cost model If an I/O is necessary the full cost of completing that operation is charged. This is not always meaningful, but can be used for example when you are completing requests that return answers of variable length.

Partial-cost model Only charge the actual number of I/O's.

0/1-cost model Charge 1 cost if an I/O is necessary, regardless the cost of completing.

Another way of evaluating algorithms is to calculate the working set (see [18]). The working set is defined as the blocks of data used within the last τ memory accesses. This method is a practical matter since it is difficult to estimate working set analytically.

In [5] it is shown that one can deduce a lower bound on the I/O-complexity when given the internal complexity.

Chapter 3

Dictionaries

A *dictionary* D is a data structure that supports three operations:

Retrieve(k,D) Locates the record with key k in dictionary D and returns the record if found.

Insert(r,D) Inserts record r, with unique key, in dictionary D.

Delete(k,D) Removes record with key k from dictionary D.

In the following we present ways of implementing dictionaries: a classical way (AVL-trees), external search trees (B-trees and Buffer trees), and hashing. Both the internal and external complexity of all three basic operations will by discussed. N is the number of records currently in a dictionary.

3.1 AVL-trees

AVL-trees was first presented by Adel'son-Velskey and Landie in [1]. An AVL-tree guarantees that the tree has a reasonable height, even at worst. This height gives an upper bound on the complexities, as shown in the following. The definition of the structure is taken from [17].

3.1.1 Definition of AVL-trees

An *AVL-tree* is a binary search tree which means that each node has at most two children, and the key of the left child is less than the key of the node, which in return is less than then key of the right child. Besides this, every node in an AVL-tree (an *AVL-node*) fulfill a property concerning the height of the node: The height of of two subtrees may never differ more than one. By a height of a node we mean the number of nodes on the longest path from the node to a leaf. This height is stored in every node.

3.1.2 The height of AVL-trees

We shall use the height of the AVL-tree to find both the internal and the external complexity of any operation on the AVL-tree.

Theorem 3.1 *The maximum height of an AVL-tree with N nodes is* $2\log_2 N + 1$.

Proof To determine the maximum height of an AVL-tree with N nodes, we ask the question in reverse, that is, what is the minimum number of nodes in an AVL-tree with height h? It is easy to see that an AVL-tree of height one has one node, and that an AVL-tree of height two has at least two nodes. For $h > 2$, the minimum number of nodes N_h in a tree with height h is determined by the following recurrence equation: $N_h = N_{h-1} + N_{h-2} + 1$. If $h > 2$, we have that $N_h > 2N_{h-2}$. Using this, N_h develops as follows

h	1	2	3	4	5	6	7	...
N_h	1	2	2	4	4	8	8	...

This means that for $h > 2$, $N_h \geq 2^{\lceil \frac{h-1}{2} \rceil}$, which leads us to the wanted result. □

Theorem 3.2 *The minimum height of an AVL-tree with N nodes is* $\log_2(N+1)$.

Proof Since the maximum number of nodes in an AVL-tree of height h is $2^h - 1$, the minimum height of an AVL-tree with N nodes is $\log_2(N+1)$. □

3.1.3 Retrieval

The retrieval procedure is the same as the retrieval procedure for a binary search tree. The procedure is shown below :

```
Retrieve(k : key, D : AVL-node)
   if D <> NIL
      if k = D.Key
         return D.record
      else if k > D.Key
         Retrieve(k,D.Right)
      else
         Retrieve(k,D.Left)
   else
      return NIL
```

Theorem 3.3 *The internal complexity of retrieving a record is* $O(\log_2 N)$.

Proof In the worst case a search proceeds to the leaf level of the AVL-tree. Since the maximum height of the AVL-tree is $2\log_2 N + 1$, and since a constant number of comparisons is made at each level, the internal complexity of retrieving is $O(\log_2 N)$. □

3.1.4 Insertion

Both insertion and deletion of a node in an AVL-tree uses an operation called *rotation*. There are two kinds of rotations: rotate left and rotate right. They are used to restore an AVL-tree that has become unbalanced by insertion or deletion. *Rotate-left(D)* rotates left on the AVL-node D and is defined as in Figure 3.1. As seen in the figure, rotate left restores balance on x when the height of y was too large compared to A. In the figure, we assume that the left subtree B of y, has a smaller height than the right subtree C of y (*single rotation*). If that is not the case, the balance is restored by first rotating right on y, and then rotating left on x (*double rotation*). These two types of rotations are both called rotate left. Rotate right are defined in the same manner. The AVL-tree is implemented

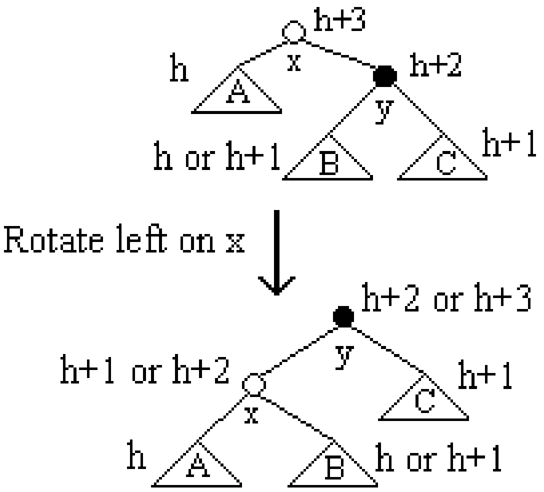

Figure 3.1: Rotate left on the AVL-node x

using pointers to store the location of children, such that each rotation can be done in constant internal time.

The insertion is then split in two sections. First, we insert the new record at the bottom of the tree, as insertion is done in a binary search tree. Secondly, we retract back along the search path, making left or right rotations at the nodes where the height difference of the children is larger than one. The algorithm is shown below:

```
Insert(r : Record, D : AVL-node)
   if D = NIL
      make D a root containing r
      D.Height = 1
   else
      if r.key < D.key
         if D.Left = NIL
            Attach r as left child of D
            D.Left.Height = 1
          else
            Insert(r,D.Left)
      else
         if D.Right = NIL
           Attach r as right child of D
           D.Right.Height = 1
         else
           Insert(r,D.Right)
      D.Height = max(D.Right.Height, D.Left.Height) + 1
      if D.Right.Height > D.Left.Height+1
        Rotate-Left(D)  {Both single and double rotations}
      if D.Left.Height > D.Right.Height+1
        Rotate-Right(D) {Both single and double rotations}
```

Theorem 3.4 *The internal complexity of inserting a record is $\theta(\log_2 N)$.*

Proof Each call of the procedure Insert has a constant internal complexity, since the rotations is done be shifting pointers to subtrees. Since the procedure is called recursively at most $2\log_2 N + 1$ times, the total internal complexity becomes $\theta(\log_2 N)$. □

3.1.5 Deletion

Deletion is done in somewhat the same way as insertion. First, the record r is located in the tree. Second, the smallest record larger than r (called x) is moved to the place of r (this is the leftmost leaf of the right subtree of x). Third, we adjust the heights and rotate along the search path of x. The algorithm is shown below :

```
Delete(k : key, D : AVL-node)
   if D.Key = k
      Find smallest record x with key larger than k
      Move x to D
      for every node from x to root
        Adjust the heights and rotate if necessary
   else
      if k < D.key
         Delete(k,D.Left)
      else
         Delete(k,D.Right)
```

Theorem 3.5 *The internal complexity of deletion is $\theta(\log_2 N)$.*

Proof No matter what record we are deleting, we always have to go to the bottom of the tree. Since the rotations take constant time, the total internal complexity becomes $\theta(\log_2 N)$. □

3.1.6 Join and split

In the next chapter we shall use AVL-trees as an auxiliary structure. For this use we expand the AVL-node to also containing the number of records in its two subtrees. Besides the three operations already described, we shall use the following two operations:

Join(a1,r,a2) Joins AVL-trees a1 and a2 with r as root. All keys in a1 are less the r's key, which in return is less than all keys in a2.

Split(A,a1,r,a2) Splits AVL-tree A returning the record r with median key of A, and AVL-trees a1 and a2. All keys in a1 are less than the key of r, which in return is less than all the keys of a2.

Splitting and joining are described in general in [39].

Theorem 3.6 *The join operations has an internal complexity of $O(\log_2 \delta)$, where δ is the height difference of a1 and a2.*

Proof a1 and a2 becomes left a right child of r. To restore the AVL-property at most δ rotations are needed. □

Splitting is divided into two parts. First, the record with the median key is sought. During the search we keep track of how many records, *N1, have smaller keys than the* *current node record, and how many records,* N2, have larger keys than the current node record. The search starts at the root and with N1 and N2 as the size of the subtrees. Otherwise if $|N1 - N2| \leq 1$ the search is finished. If $N1 > N2$, the search is continued along the left subtree. If $N2 > N1$, the search is continued along the right subtree. Second, the two new AVL-trees a1 and a2 are created. Initially, a1 and a2 are the left and right subtree of the median node. The creation starts at the median node just found and traverses to the root of the AVL-tree. If a step up in the AVL-tree goes to the left (towards the records with smaller keys) then a1 becomes the right subtree of the node we came to and that node becomes the root of a1. To make sure that a1 is an AVL-tree after this, we check the heights and rotate if necessary. Likewise if we go to the right.

Theorem 3.7 *The split operation on an AVL-tree with N records has an internal complexity of* $O(\log_2 N)$

Proof Finding the record with the median key has an internal complexity of $O(\log_2 N)$. Since each join during the traversal from the record with the median key to the root has internal complexity $O(1)$, and the height of an AVL-tree is $O(\log_2 N)$, the total internal complexity becomes $O(\log_2 N)$. □

3.1.7 External complexities

Since the AVL-tree uses pointers and rotations take place during insert and deletion, the AVL-nodes are not placed systematically corresponding to the position within the AVL-tree. This means that, at worst, accessing a node causes one block transfer. Therefore the external complexity of all three operations is equal to the corresponding internal complexities. As we shall see, the dictionary problem can be solved with the same internal complexity and better external complexity.

3.2 B-trees

B-trees where introduced by Bayer and McCreight in [9]. They have a reputation for providing fast implementation for dictionaries. In [9] the internal complexity is not considered, but we calculate it to be $O(B \log_B N)$ per insert or delete operation and $O(\log_2 N)$ per retrieve operation. We choose an internal structure which gives an internal complexity of $\theta(\log_2 N)$ per insert and delete operation and $O(\log_2 N)$ per retrieve operation.

3.2.1 Definition of B-trees

B-trees are (a,b)-tree with a=$\lfloor \frac{B}{4} \rfloor$ and b $= B$ (see [33] for a definition of (a,b)-trees). In order to define *B-trees*, we begin by defining a node in a B-tree. A *B-node* has room for at most B records inserted in an AVL-tree, and B+1 pointers to other B-nodes. For simplicity we assume that records and pointers of a B-node fit in a B-block. In practice the number of stored records would be less. A B-tree is then defined as a tree of B-nodes with maximum fanout of B+1. The following conditions are to be fulfilled by a B-tree :

- The number of nodes on each path from the root to any leaf have the same length, which we call the *height* of the B-tree.
- Each B-node except the root and the leaves has at least $\lfloor \frac{B}{4} \rfloor + 1$ children. The root is either a leaf or has at least two children. The leaves contain a least $\lfloor \frac{B}{4} \rfloor$ records.
- Each B-node has at most B+1 children (that is B records).

- The keys of the records of B-node Q partition the descendents of Q in intervals.

In the following figures, B-trees are drawn as if the records in the B-nodes are in a sorted list and not in an AVL-tree. This is done to make the figures easier to read. Figure 3.2 is a B-tree.

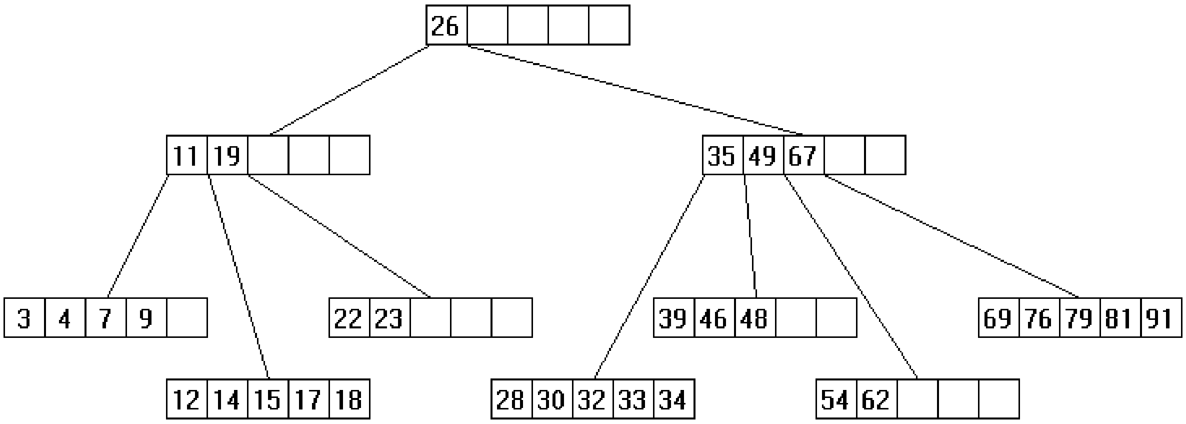

Figure 3.2: A B-tree with B = 5

3.2.2 The height of B-trees

In order to calculate the internal and external complexities of operations on B-trees, it is useful to know the height of a B-tree with N records.

Theorem 3.8 *The height of a B-tree with N records is* $\theta(\log_B n + 1)$.

Proof A B-tree with N records has at most $\lceil \frac{4N}{B} \rceil$ nodes. Since the degree of a B-tree is at least $\lfloor \frac{B}{4} \rfloor$, the height is $\log_{\lfloor \frac{B}{4} \rfloor} \lceil \frac{4N}{B} \rceil + 1 = \theta(\log_B n + 1)\Box$

3.2.3 Retrieval

The retrieval algorithm is similar to the retrieval algorithm of a binary search tree. The algorithm is shown below :

```
Retrieve(k : Key, D : B-Node)
   if D <> NIL
      temp = AVL-retrive(k,D.AVL)
      if temp = NIL
         Select the appropriate child P of D by retrieving
             in the AVL-tree.
         Retrieve(k,P)
      else
         return temp
   else
      return NIL
```

Theorem 3.9 *The internal complexity of retrieving a record is* $O(\log_2 N)$.

Proof The internal complexity of deciding whether a record is in a B-node and that of selecting the appropriate child of D is $O(\log_2 B)$. Since the height of a B-tree is $\theta(\log_B n + 1)$, the total internal complexity is $O(\log_2 B(\log_B n+1)) = O(\log_2 n+\log_2 B) = O(\log_2 N)$. □

Theorem 3.10 *The external complexity of retrieving a record is* $O(\log_B n)$.

Proof The external complexity of the work done in each B-node is $O(1)$ since the B-node has size B. Since the height of a B-tree is $\theta(\log_B n + 1)$, the total external complexity is $O(\log_B n)$.□

Later we shall use the internal complexity of retrieving a leaf B-node, therefore the following theorem:

Theorem 3.11 *The internal complexity of retrieving a leaf B-node is* $\theta(log_2 n)$.

Proof This operation is the same as retrieving a record in a leaf except we do not make the AVL-tree retrieval in the leaf B-node. The complexity then becomes $\theta(\log_2 N - \log_2 B) = \theta(\log_2 n)$. □

3.2.4 Insertion

Insertions in B-trees are always made at the leaves. First, we decide into which leaf the record is to be inserted. This is done as in retrieving. If the leaf contains less than B records the record is inserted in the AVL-tree. If the leaf contains B record we have to perform a process called splitting. *Splitting* takes the leaf and splits it into two new B-nodes. First, we move the AVL-tree to a new location in the internal memory. Second, we insert the new record in the AVL-tree. Third, we split the AVL-tree as described in Section 3.1.6. Fourth, two new B-nodes are created and the two AVL-trees from the split are moved there. The two new nodes becomes leaves in the B-tree. The median record of the old leaf is recursively inserted in the parent which may be split. If the root of the B-tree has to be split, a new root containing one record is created. Since the length of the path from the root to a leaf is only changed when the root is split, the length of the path from each leaf to the root remains equal during insertion. Figure 3.3 illustrates the algorithm.

The algorithm is shown below:

```
Insert(r : record, D : B-Node)
   Find leaf L to contain r
   if size(L) <= B
      AVL-tree-Insert(r,L.AVL)
   else
      Split(r,L)

Split(r : record, B : B-Node)
   Make two B-nodes B1 and B2
   move B.AVL to new location
   AVL-tree-insert(r,B.AVL)
   AVL-Split(B.AVL,a1,mid,a2)
   move a1 to B1
   move a2 to B2
```

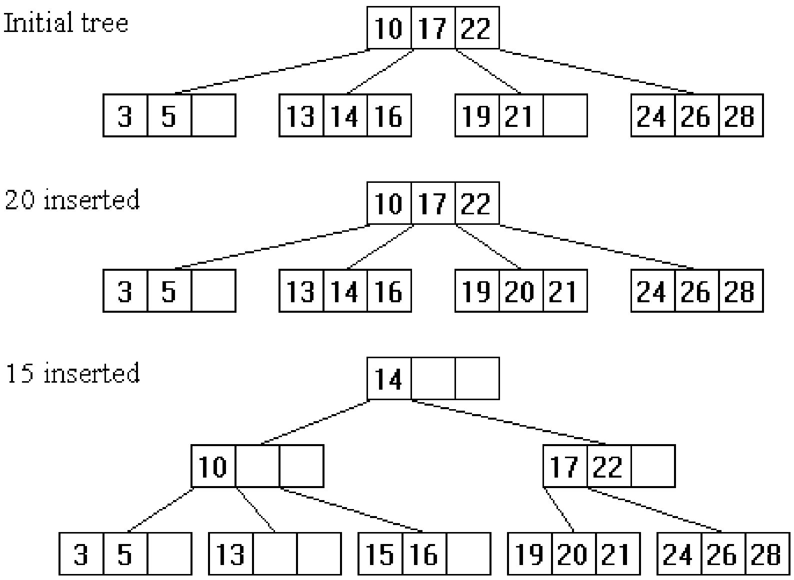

Figure 3.3: Illustration of B-tree insertion

```
if B = Root
   insert mid in new B-Node
   make the new B-node the root
else
   if size(B's parent) <= B
     AVL-tree-insert(mid,B.parent.AVL)
   else
     Split(mid, B's parent)
discard(B);
```

Theorem 3.12 *The amortized internal complexity of insertion is $\theta(\log_2 N)$.*

Proof As shown in Theorem 3.11, the internal complexity of finding the right B-leaf is $\theta(\log_2 n)$. Inserting the record in the AVL-tree has an internal complexity of $\theta(\log_2 B)$ giving a total of $\theta(\log_2 N)$. As the splitting returns two half full B-nodes and is only made when a B-node is full, the splitting is only made at most every $\frac{B}{2} = O(B)$ times. As we shall see later, the deletion does not affect this. Splitting an AVL-tree has an internal complexity of $O(\log_2 B)$. Moving the AVL-trees from one B-node to another has an internal complexity of $\theta(B)$. Since we can be forced to insert a record at each level of the B-tree (we split at each level), there are at most $O(\log_B n + 1)$ splits. The splitting then has an internal complexity of $O(\frac{(\log_2 B+B)(\log_B n+1)}{B}) = O(\log_B n + 1) = O(\log_B N)$. The total internal complexity then becomes $\theta(\log_2 N) + O(log_B N) = \theta(\log_2 N)$. $\square$

Theorem 3.13 *The amortized external complexity of inserting a record is $\theta(\log_B n)$*

Proof AT worst we access a constant number of blocks on each level of a B-tree during insertion. We get the wanted result from the fact that the height of a B-tree is $\theta(\log_B n+1)$. $\square$

3.2.5 Deletion

Deletions are a bit more complicated then insertions. First, we have to define what we mean by 'stealing' a record from a brother. B-node P, can steal a record from its right brother, by moving the smallest record larger than all the records in P, from P's parent to P, and then moving the smallest record from P's right brother to P's parent, replacing the record just moved. Last, we move the corresponding pointer in P's right brother to P, placing it after the new record in p. We define stealing from P's left brother similarly. Figure 3.4 illustrates this operation.

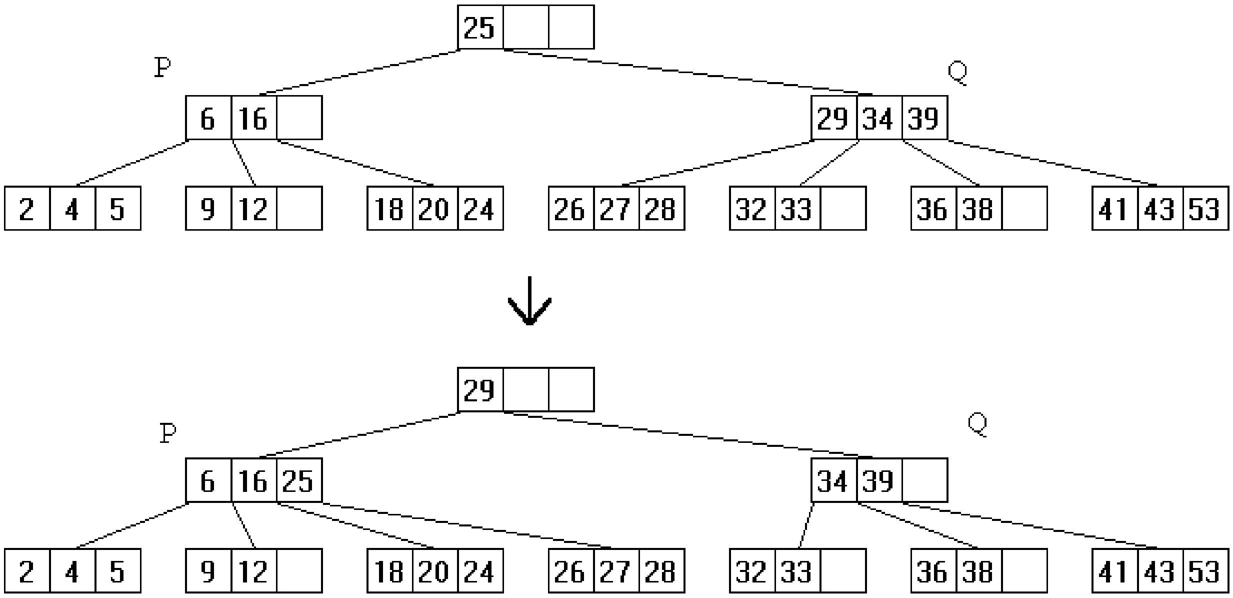

Figure 3.4: P steals a record from Q

In order to delete a record from a B-tree, we first locate the record. The deletion is then divided into two cases :

1. **The record is located at a leaf :** First we remove the record from that leaf. If, after the removal, the leaf contains less than $\lfloor \frac{B}{4} \rfloor$ records, we have to do some stealing. If the right brother contains more than $\lfloor \frac{B}{4} \rfloor$ records, we steal a record from there. If not, we check the left brother. If none of the brothers contain enough records, we make a new B-node, melt the leaf (containing $\lfloor \frac{B}{4} \rfloor - 1$ records), one of the brothers (containing $\lfloor \frac{B}{4} \rfloor$ records), and the record from the parent of the leaf falling between the leaf and its brother. We can use the join operation described in Section 3.1.6 to join the two AVL-trees. The new node will contain $\frac{B}{2}$ records. If the parent contains fewer than $\lfloor \frac{B}{4} \rfloor$ records after the above operation, we check its brothers for something to steal. If we cannot steal anything, we melt like before. The operation is propagated up the tree. If the root has only two children and these children are melted, the root is discarded, and the melted children become the new root.

2. **The record is located in an internal node or in the root :** The deletion is made by copying the smallest record larger than the record we want to delete from the leaf to the node, overwriting the record we wanted to delete. The smallest larger

record is located at the leftmost leaf of the right subtree. The copied record is then deleted as described above.

Since the height of any B-node is only changed when the root has only one record, and the two children are melted, the length of the path from the root to a leaf will always remain the same for all the leaves. Figure 3.5 illustrates the algorithm. The algorithm is shown

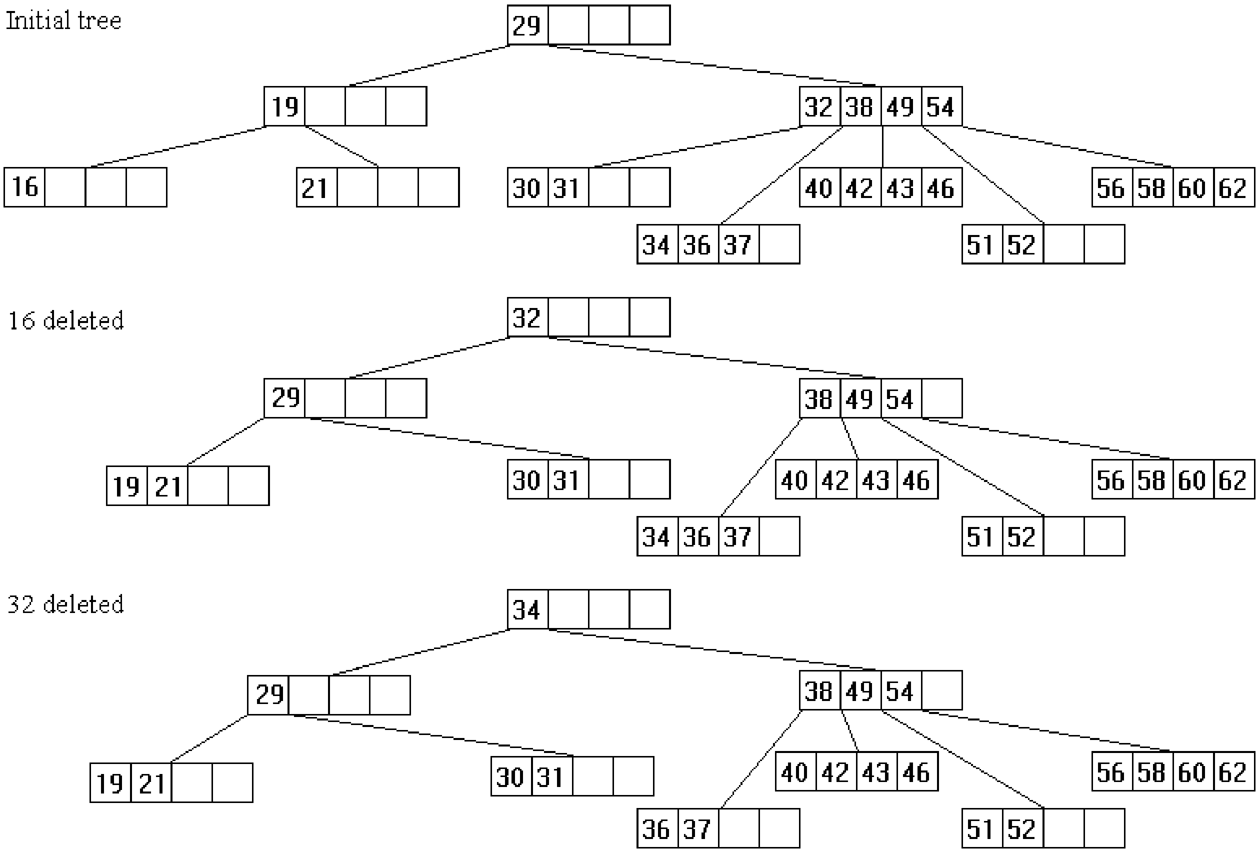

Figure 3.5: Illustration of B-tree deletion

below :

```
Delete(k : key, D : B-Node)
   Find the node L containing k
   if L is leaf
      DeleteLeaf(L,k)
   else
      DeleteInternal(L,k)

DeleteLeaf(D : B-Node, k : key)
   AVL-tree-delete(k,D.AVL)
   CheckSize(D)

DeleteInternal(D : B-Node, k : key)
   locate the record r with smallest key larger than k
   copy r to D overriding k's record
```

```
   DeleteLeaf(node containing r , r.key)

CheckSize(D : B-Node)
   if D = Root
      if size(D) = 0
         discard root and make D's child new root
   else if size(D) < B/4
      BR = D's right brother
      if size(BR) > B/4
         steal from BR
      else
         BL = D's left brother
         if size(BL) > B/4
            steal from BL
         else
            make a new node BN
            BN.AVL = AVL-tree-melt(BR.AVL,dividing record from D's parent,BL.AVL)
            CheckSize(D's parent);
```

Theorem 3.14 *The amortized internal complexity of deleting a record* $\theta(\log_2 N)$*.*

Proof If the record we want to delete is residing on a leaf of the B-tree, we retrieve the B-leaf on which it is residing, and deletes the record from the AVL-tree. This has a total internal complexity of $\theta(\log_2 n + \log_2 B) = \theta(\log_2 N)$ (see Theorem 3.11). If the record is residing in an internal B-node, the record is found and thereafter the smallest record with larger key is found and deleted ($\theta(\log_2 n + \log_2 B) = \theta(\log_2 N)$). This means that before calling DeleteLeaf, we have used $\theta(\log_2 N)$ time, the same time as if the record we wanted to delete were on a leaf. As the melting operation returns a B-node with $\frac{B}{2}$ records and is only made when a B-node contains less than $\frac{B}{4}$ records, the melting is only made at most every $\frac{B}{4} = O(B)$ times. Since both the melting and the spliting (see section on inserting in B-trees) operations return B-nodes containing $\frac{B}{2}$ records, deletion and inserting do not affect each others complexities. The recursive CheckSize procedure, could at worst melt B-nodes on each level of the B-tree. The height difference of the two AVL-trees is at most $O(\log_2 B)$ the melting can be made in $O(\log_2 B) + \theta(B) = \theta(B)$ time (B comes from moving an AVL-tree from one node to another) and there are at most $O(\log_B n + 1)$ melts, the total complexity for CheckSize is $O(\frac{B(\log_B n+1)}{B} = O(\log_B n + 1)) = O(\log_B N)$. The total internal complexity then becomes $\theta(\log_2 N) + O(\log_B N) = \theta(\log_2 N)$. □

Theorem 3.15 *The amortized external complexity of deleting a record is* $\theta(log_B n)$*.*

Proof At worst we access a constant number of blocks on each level of a B-tree during deletion. We get the wanted result from the fact that the height of a B-tree is $\theta(\log_B n+1)$. □

3.2.6 The optimal B-node size

In the section we will show that the optimal B-node size is B, that is a B-node with B records and $B + 1$ pointers. In the following we will assume that a B-node has size T. The internal complexity of every AVL-tree operation then becomes $O(\log_2 T)$. The height of the B-tree is $\theta(\log_T N)$ which leads us to an internal complexity for every operation of $\theta(\log_2 T \log_T N) = \theta(\log_2 N)$. It is clear that the external complexity of the AVL-tree operations in each B-node is $O(\lceil \frac{T}{B} \rceil)$. This gives us an external complexity of $\theta(\lceil \frac{T}{B} \rceil \log_T N)$

which is shown in Appendix A Proposition A.2 to have a minimum at $T = B$. The conclusion is that the optimal B-node size is B.

3.2.7 B-trees variants

We will briefly describe three variants on the basic B-tree scheme, namely B*-trees, B$^+$-trees and prefix B$^+$-trees. There seems to be some confusion on the use of the term B*-tree, but we adopt the usage presented by D. Comer in [14].

B-trees* are a B-tree where the B-nodes are at least $\frac{2}{3}$ full (instead of the usual $\frac{1}{4}$ full). B*-tree insertion uses an local distribution scheme to delay splitting, that is insertion in a full B-node causes a movement of a record from the B-node to one of its brothers. If the brother is full, the two B-nodes are divided into three each of size $\frac{2}{3}$. B*-tree deletion 'steals' from its brother node and its brothers brother node until all three B-nodes are $\frac{2}{3}$ full. Melting these three nodes gives two full B-nodes. The B*-tree has a storage utilization of at least 66 percent and, as a side effect, the fanout becomes large giving a tree with less height.

In *B$^+$-trees* all records reside on the leaves. The upper levels, which are organized as a B-tree, only have copies of the keys, making it possible to locate any record. The advantages of B$^+$-trees are many. First, the keys take up less storage than the records, giving a bigger fanout of the B-nodes. Second, as it is not necessary to delete keys from the internal B-nodes during deletion of a record, the process is simplified. The keys in internal B-nodes are only deleted when a leaf becomes less than $\frac{1}{4}$ full. Third, it is easy to link all records together, giving an easy way to implement a next or previous operation.

The *prefix B$^+$-trees* are a further development of the B$^+$-trees. The keys in the internal B-nodes are replaced by a prefix of these keys. The smallest possible prefix splitting the underlying records is chosen. For example the keys [151, 167, 189, 212, 219] can be split in the two parts, [151, 167, 189] and [212,219], by the prefix 1 instead of the key 189. The prefix B$^+$-trees therefore have more keys in each internal node than the B$^+$-trees, giving a bigger fanout.

3.3 Buffer trees

A new external dictionary structure was recently presented by L. Arge [6], named a buffer tree. A *buffer tree* is an (a, b)-tree with $a = \frac{m}{4}$ and $b = m$, extended with a buffer of m blocks at each internal node. Each leaf is a B-block. Figure 3.6 shows an illustration of a buffer tree. When performing a delete or an insert operation on the structure, a buffer record is constructed containing the operation type, the record to operate on and a time stamp. The buffer record is then inserted in the roots buffer. If the buffer is now larger than $\frac{m}{2}$ blocks, the buffer is split according to the partitioning records at the root and pushed down to the child nodes. After the push down of a new buffer, this new buffer is merged with the buffer already at the node if the extended buffer is larger than $\frac{m}{2}$, it is split and pushed downwards. While we are merging the two buffers, the delete and insert operations annihilate if the operations match according to record and time-stamp. For example, a new deletion of a record removes any old operation on the same record. If the children of a buffer-pushing node are leaves, a split or fusion of the parent node might occur. A node split happens if the number of children are greater than m (because of inserted records). A fusion occurs when the number of children become lower than $\frac{m}{2}$ (because of deletions). The splits and fusions can propagate upwards.

Arge showed that the I/O-complexity of an arbitrary sequence of N intermixed insertions and deletions has an amortized complexity of $O(\frac{1}{B}\log_m n)$. Note that N denotes the

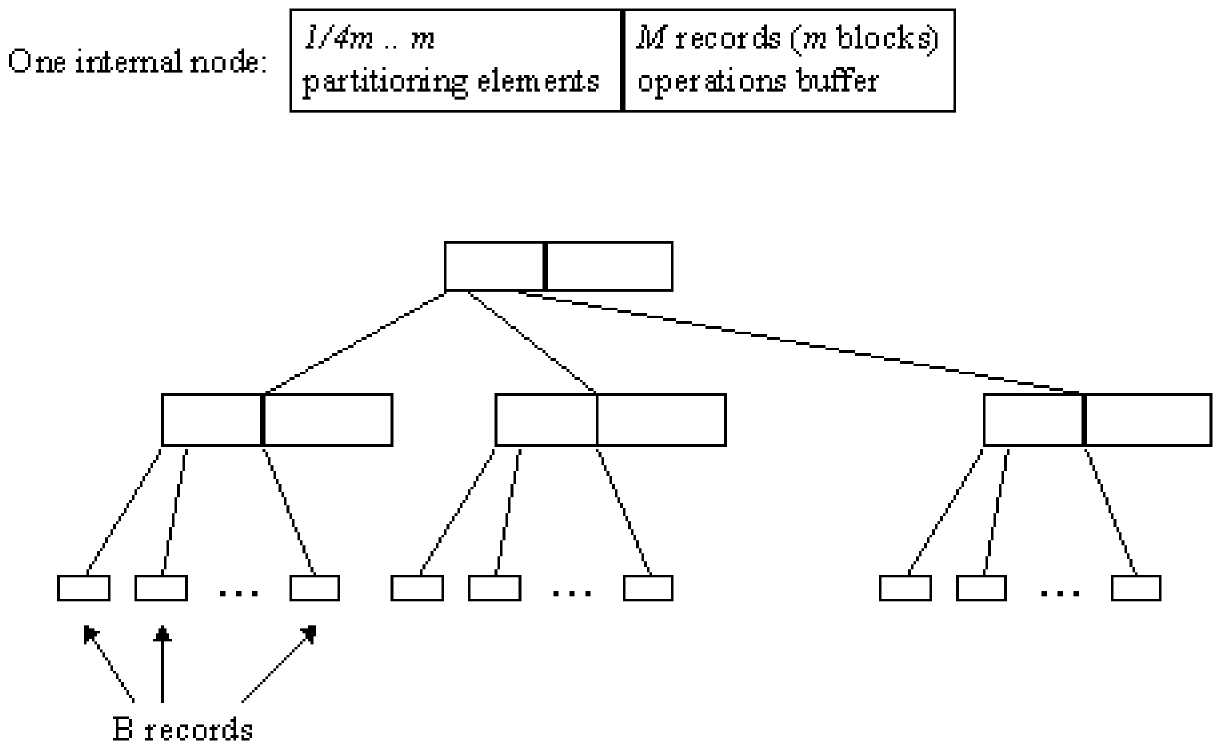

Figure 3.6: Illustration of buffer tree

number of operations and not, as we usually use it, the number records currently in the data structure.

Retrieving a record is not done 'real-time' since the operations (also retrieving) are buffered. This makes the structure less useful than other dictionaries, but can be used in certain application where the operations are known in advance. Arge used the structure for developing an I/O-optimal algorithms for some geometric problems.

3.4 Hashing

In hashing, records are mapped into an array. The keys of records are interpreted as inputs to a mapping function, which as output has array positions. The problem of hashing is that the mapping might place two different records in the same position. This conflict can be solved in many ways. We discourse two of these approaches namely *chaining* and *open addressing*. Besides these two, designed as internal algorithms, we examine two external algorithms *extendible hashing* and *external hashing*.

The size of the array, S, is known at all times, but may in some implementations change during the operations. To keep a proportion between the number of records in the dictionary and the size of the dictionary, an array is doubled or halved (and rebuilt) according to the number of records inserted. The doubling or halving of a hash table with N records inserted will mean N insertions when rebuilding. When doubling, one could simply keep the current hash function (and not move the records), but then the records would not be evenly spread over the hash table, which is important for an efficient execution of the dictionary. The N insertions when rebuilding are covered by the next $\theta(N)$ operations (conceptually), since a doubling or a halving can only occur after every $O(N)$ operations respectively. As we will see below this supplement to each of the following operations does not effect the complexity of the different operations.

3.4.1 Hash functions

A good hash function maps the keys of the records to be inserted uniformly, but choosing only one function has the risk of doing bad for a given set of operations. A way out of this dilemma is to use *universal hashing*, where one chooses a class of hashing functions and not a single hash function when designing a hash algorithm. When the hash algorithm is actually used, a hash function is picked randomly. This has the advantage that the execution is not dependent on one choice but rather on a random choice which changes at every usage. One can now guarantee a good *average* performance. An example of a class of functions is [33]: $H_1 = \{h_{a,b} | h_{a,b} = (ax + b) \bmod N) \bmod S, a, b \in [0..N-1]\}$ where N is a prime.

3.4.2 Chaining

Collisions are handled by viewing each mapped position as a list of records, ie. the records with the same array position are inserted in the same list, see figure 3.7. Inserting a record

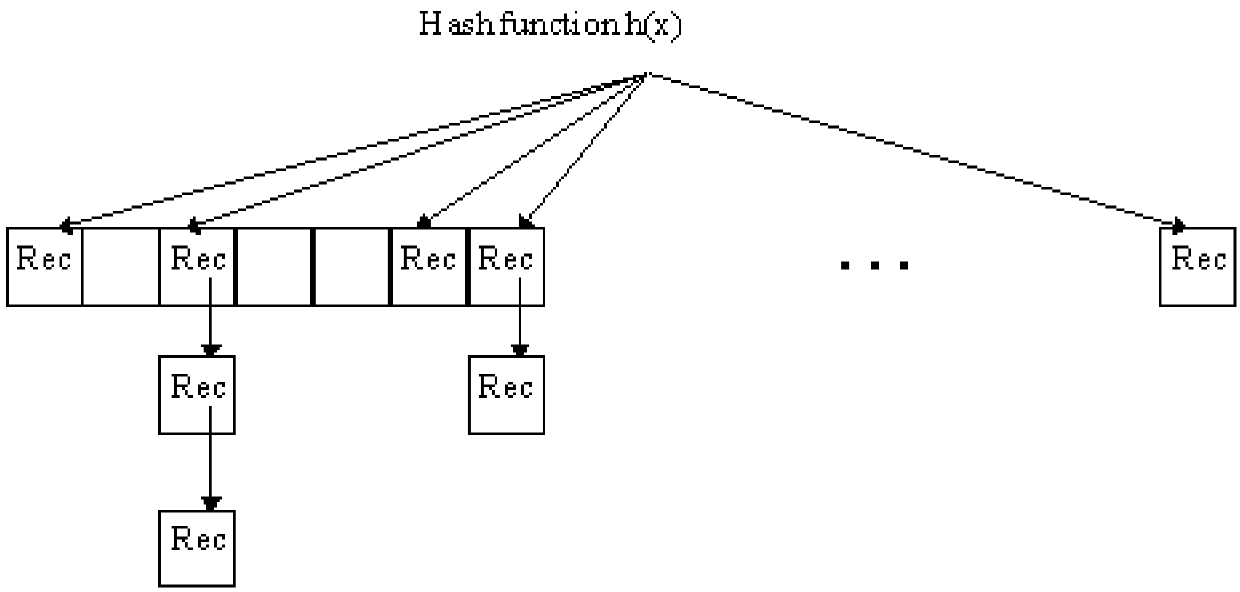

Figure 3.7: Hashing using chaining

is made in constant time, by simply placing the record as the first or the last record in the proper list. On the other hand, retrieving and deleting a record can at worst take linear time if one list contains many of the records, but when you assume universal hashing the following can be shown [33], when using for example the function class H_1 defined in Section 3.4.1:

Proposition 3.1 *When you use hashing with chaining and you assume universal uniform hashing, the following holds:*

- *The expected cost of retrieve, insert or delete operation is* $O(1 + c\beta)$
- *The expected cost of a sequence of* N *retrieval, insert and delete operations starting with an empty dictionary is* $O(1 + \frac{c\beta}{2})N)$

where $\beta = \frac{N}{S}$ *is the (maximum) load factor and* $c \in \Re$*.*

When you use the function class H_1 c is almost 1, and since β will be less than 1 ($N \leq S$), the internal complexity for all operations is $O(1)$. The external complexities are also $O(1)$, since each operation requires $O(1)$ access to external memory. As mentioned above, the costs of rebuilding are $\theta(N)$ insertions, which are amortized as a cost to the previous $\theta(N)$ operations. Since one insertion has a complexity of $O(1)$, like the other operations, the supplement of the cost of rebuilding does not increase the complexity of the operations.

3.4.3 Open addressing

Another way of solving collisions is by open addressing. This method keeps generating array positions until a vacant slot is found. This is also called probing. Each key value K is said to have a probe sequence determined by K. The probes can be found in several ways, see for example [23], one way is to use the current probe (as input to a function) to calculate the next probe (see [33]), see figure 3.8. However deletions are difficult to implement because positions cannot be freed without disturbing the generation of the array positions. The table ends up being full of unused slots, if deletions are allowed. The internal and external complexities are equal because the generated array positions

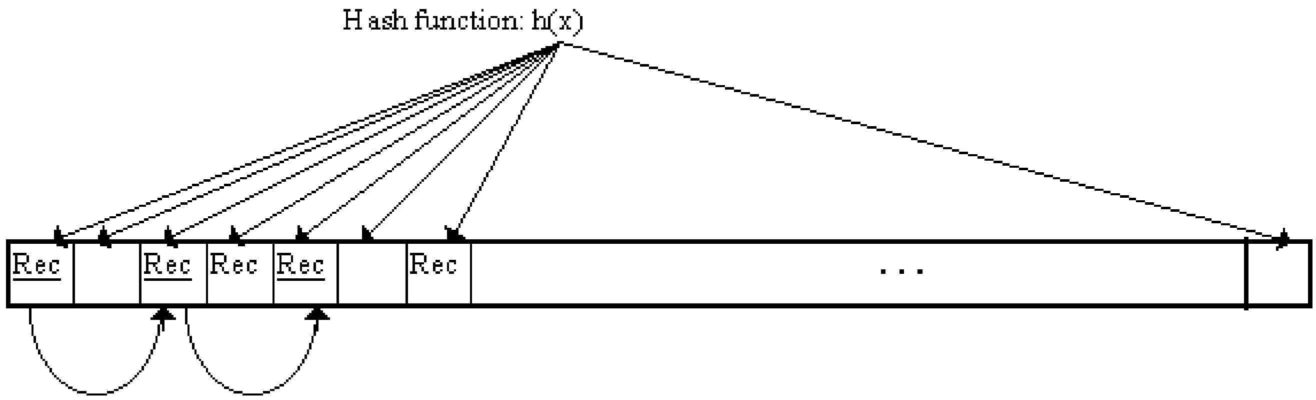

Figure 3.8: Hashing using open addressing

might require an I/O to check whether the position is empty. In [33] it is shown that the expected cost of an insertion is $O(\frac{1}{1-\beta})$ where $\beta = \frac{N}{S}$. A retrieval has an expected cost $O(\frac{1}{\beta} \ln \frac{1}{1-\beta})$. Only doubling is carried out when you use open addressing, since we assume that deletions are not performed. It is clear that $O(N)$ insertions cover the cost of the next rebuilding, if the structure contains N records prior to rebuilding.

3.4.4 Extendible hashing

Hashing with chaining and open address hashing mapped the records directly into a hash table and, if necessary, further on to lists rooted in the hash table. In the following we present a method, *extendible hashing* that maps the hash function to a *directory* which in return maps records to the B-blocks of the external memory. The output of the hash function is interpreted as strings of bits. The directory is a structure which during execution can be doubled or halved. Each *slot* of the directory points to a B-block, where several slots can point to the same block. The directory has a current depth d, indicating that the current size of the directory is 2^d's slots. At first the depth is 1 and the directory has two slots. The slots are identified by the 2^d different permutations of d bits. To locate which B-block a record with key x belongs to, the following steps are executed:

- calculate the hash function value $h(x)$

- extract the d first bits of $h(x)$
- locate the slot identified by the d bits and return the pointer of the slot

In the following example (Figure 3.9) the depth is 2, the directory has four slots, and five records are inserted. Even though the directory has four slots only three block are in use, each block can contain two records ($B = 2$). Inserting a record is split in two

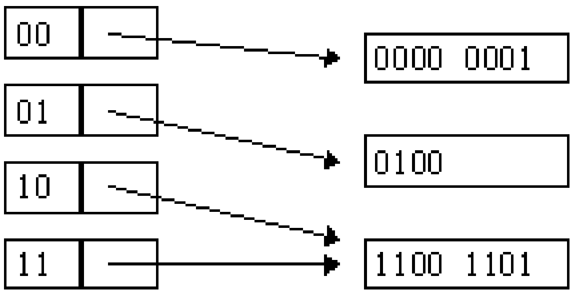

Figure 3.9: Five records inserted with the hash function values: 0000, 0001, 0100, 1100, 1101

cases. In the first case the located block is not full and the record is simply placed in the block. In the second case, the located block is full and a new block is needed. If the new block is fitted into the directory, that is if the new block can be pointed out by one of the slots which is not in use, the directory is not doubled. This will be the case in the former example when inserting a record with hash value 1000 (see Figure 3.10). The slot with identification 10 is used to point out the new block and the new record is inserted. If on

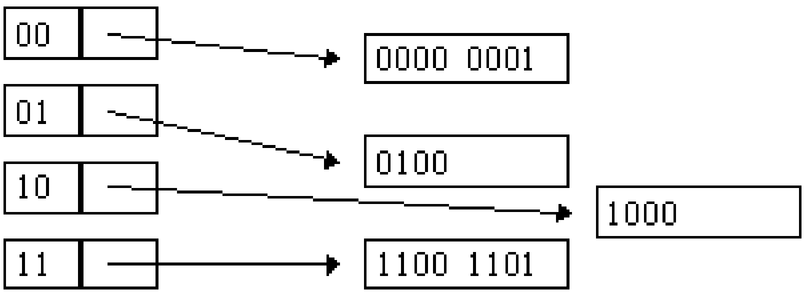

Figure 3.10: Six records inserted with the hash function values 0000, 0001, 0100, 1000, 1100, 1101

the other hand the directory is not deep enough to separate the hash values, the directory is doubled. This is done by first increasing the depth by one, then doubling the number of slots and finally doubling the pointers so that each block in the old structure is pointed out twice the times it was before. The new block can now be fitted into the directory, as was the case with the former example. The doubling is shown in the following example where we insert a record with hash value 0010 into the structure of the first example, see Figure 3.11. Deletion is made in a similar manner but with merging of B-blocks and halvings of the directory if necesarry.

In [33] the following properties are shown with reference to [21]:

Theorem 3.16 *The expected number of B-blocks required to store N records is approximately $\frac{N}{B}\ln 2$. The expected size of the directory is approximately $(\frac{e}{B}\ln 2)n^{1+\frac{1}{B}}$*

Since the directory cannot be kept in internal memory, a locating of a block requires one block read. Retrieving a record will cost two block reads. An insertion or deletion

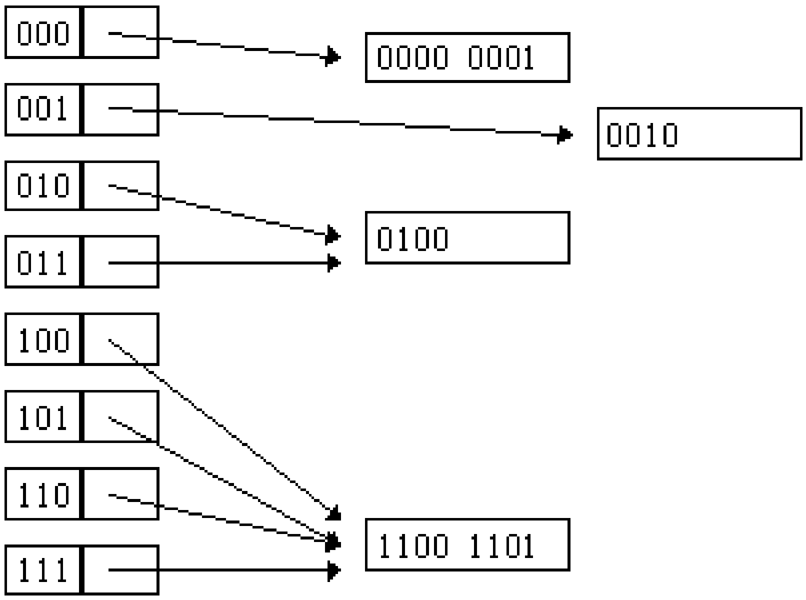

Figure 3.11: Six records inserted with the hash function values 0000, 0001, 0010, 0100, 1100, 1101

will always cost at least two block reads and one block write. When considering the costs of directory changes, the I/O-complexities of insertion and deletion are $3 + \epsilon(B)$.

3.4.5 External hashing with limited internal memory

In Section 3.4.3 the probing operation resulted in a block read for every probe. But utilizing the internal memory can reduce the external complexity of probing. Gonnet and Larson [24] present algorithms which use internal memory to store information concerning the location of records in the external B-blocks. Three algorithms (or schemes) for arbitrary block sizes are described, the first algorithm assumes that the amount of internal information per block is variable. We discard this solution since we cannot guarantee that the additional information can be fitted into internal memory. The second and third schemes use a fixed amount of internal information per block, this assumption forces a maximum on the number of blocks that can be allocated when having a limited internal memory. Gonnet and Larson do not study the second scheme in details since they find the third algorithm a more practical and interesting choice.

The performance of the algorithm is a trade-off between three variables (two are fixed while the third is variable):

retrieval speed expected I/O-complexity of insertion, retrieval is guaranteed to be one block read.

internal storage number of bits per external B-block

external storage load factor α of the B-blocks

Besides the hash function a *hash signature function* is assumed. The signature function generates a signature per probe determined by the key of a record, and runs parallel to the probing sequence. A signature is interpreted as a bit representation of a number in the range $[0, 1[$. A *separating prefix* is a bit pattern which separates a set of bit strings (in our case signatures) in two groups. The algorithm finds a separating prefix separating a set of sorted signatures obeying two restrictions: the size of the found prefix is d and make the size of one group as close to B as possible but $\leq B$. The d bits of internal information per block codes for one of the three following situations:

- The block is empty.
- The block is occupied but has not been *overflowed* (no more than B records have probed this block). A bit string representing the signature 1 is stored.
- The block has been overflowed. A bit string is stored, separating the signatures of the records probing this block in two.

When a block is overflowed, the records are divided so that the separating prefix has size d. The best division (so that one block is as close to B as possible) is found as follows:

- Calculate and sort the signatures of the records in the block and the record causing the overflow.
- If the prefixes (of length d) of the Bth and the $B-1$th record are equal, we try the next pair and so on, until we find a pair with different prefixes.

Notice that the separating prefix can only decrease (in the represented value) after every overflow operation, this ensures that records separated in an earlier separation are still excluded appropriate. The location of record can now be made without any external accesses, since the location operation can decide whether or not a record belongs to a block during a probe sequence by comparing with the separating prefix of that block. Retrieval is as mentioned guaranteed just one block access. The insertion costs are found as an expected cost, but is a complicated expression based on numerical analysis to get a result. Table 3.1 shows how many pages are modified with $\alpha = 0.8$ and some example parameters. To calculate the number of I/O's you double these numbers (one block read

	$B = 10$	$B = 20$
String size= 4	1.6721	1.3602
String size= 8	1.5119	1.2660

Table 3.1: Pages modified during insertion.

and one block write per modification).

3.5 Related work

Our work has been concentrated on structures supporting one-dimensional queries, i.e. the query has one-dimensional answers. In [7] an optimal interval structure is presented. Besides the article has a section which lists previous results on the area.

In [12] practical experiments are carried out on three different dictionaries: B-trees, 234-trees and 234-trees-core.

3.6 Conclusion

Table 3.2 summerizes the performance of the various dictionary structures. Empty fields mean that the costs are not calculated, where as fields containing a -, mean that the

		AVL-trees	B-trees	Chaining	Open addressing	Extendible hashing	External hashing
Retrieve	internal	$O(\log_2 N)$	$O(\log_2 N)$	$O(1)$	$O(\frac{1}{\beta}\ln\frac{1}{1-\beta})$		
	external	$O(\log_2 N)$	$O(\log_B n)$	$O(1)$	$O(\frac{1}{\beta}\ln\frac{1}{1-\beta})$	2	1
Insert	internal	$\theta(\log_2 N)$	$\theta(\log_2 N)$	$O(1)$	$O(\frac{1}{1-\beta})$		
	external	$\theta(\log_2 N)$	$\theta(\log_B n)$	$O(1)$	$O(\frac{1}{1-\beta})$	$3+\epsilon(B)$	2.5320
Delete	internal	$\theta(\log_2 N)$	$\theta(\log_2 N)$	$O(1)$	-		-
	external	$\theta(\log_2 N)$	$\theta(\log_B n)$	$O(1)$	-	$3+\epsilon(B)$	-

Table 3.2: Complexities of various dictionary structures.

operation is not possible. AVL-trees are a bad choice because the height of the structure is large, the structure is often balanced and finally the use of pointers makes the memory accesses arbitrary. B-trees have rather good complexities and seems much better than AVL-trees. Hashing has the best complexities. One might choose B-trees instead, because of the worst-case complexities, whereas the hashing has expected complexities. External hashing seems be the best choice, but uses extra internal storage, and does not support deletions. Buffer trees are not compared since they can not be used for real-time retrieval.

Chapter 4

Priority queues

A priority queue is a data structure that supports the following operations:

insert(r, P) Inserts record r with an arbitrary priority in priority queue P

delete-max(P) Returns the record in P which currently has the highest priority

For the sake of simplicity we will assume that a high priority means a high key value (the deletions are therefore delete-max and not delete-min). In some applications it is furthermore useful to support upgrading of a priority *increase-key*(r-id, k, P), which given a record id r-id and a new key value k, upgrades the key value of the record i P with record id r-id.

In this chapter N usually denotes the current size of a queue and n the number of blocks currently occupied by the queue. When B is the size of the blocks in records $N = cBn$ for some constant c depending on the structure in question.

We will define an I/O-optimality which is not general. This is based on the lower bound of sorting (both internal and external complexity). We call a priority queue I/O-optimal, if using the insert and delete-max operations, one can sort matching the lower bounds of sorting. When working with the comparison based model the lower bound on the internal complexity is $\Omega(N \log_2 N)$ (see for example [30]) when sorting N records. This means either that the execution of an insertion or delete-max (or both) must be $\Omega(\log_2 N)$. Analogously one can deduce a lower bound on the external complexity. Aggarval and Vitter [4] showed that the lower bound on the I/O-complexity of sorting is $\Omega(nlog_m n)$. From this we can conclude that either the execution of an insertion or delete-max must require $\Omega(\frac{1}{B} \log_m n)$.

In this chapter we present the classical solution (d-heaps), an external structure (B-trees) and finally two I/O-optimal solutions (the I/O-queue and the Buffer tree).

4.1 d-heaps as priority queues

The description of d-heaps can be found from many sources [10, 16, 31, 43].

4.1.1 The data structure

A *d-heap* consists of records arranged in a tree structure with a fan-out of d. To ensure an optimal height the structure is kept complete, meaning that all internal nodes except the rightmost must have d children. The rightmost internal node and righmost leaf is defined below. The records are arranged so that the following heap-condition is valid: the children of a node do not have greater key values than the node itself. This means that the root of the tree will contain a record with maximum key value.

Now the relationship between N and n is simply $n = \lceil \frac{N}{B} \rceil$. When storing N records the height of the heap is $h = \lfloor \log_d(N-1) \rfloor + 1$ for $N > 1$, where *height* denotes the number of nodes of a path from the root to any node. For $d > B$, the height can also be expressed as $h \le \log_d N + 1 \le \log_d \frac{N}{B} + 2 \approx \log_d n + 2$. The *level of a node* denotes the number of nodes from the node to the root. The root is therefore at level 1, while the leafs are at levels h and $h-1$.

The records are stored in an array and in general one can calculate the location of the parent and the children for any node in a d-heap as follows (locations from 0 to N-1):

$parent(x) = \lfloor \frac{x-1}{d} \rfloor$, for $x \in \{1 \ldots N-1\}$

$children(x) \in \{dx+1, \ldots, d(x+1)\} \cap \{0, \ldots, N-1\}$

Note that the children of a node are placed continuously. Meaning that when all the child nodes are touched, at most $\lceil \frac{d}{B} \rceil + 1$ blocks are read. The *righmost leaf* is the N'th record and the *righmost internal node* is the parent of this leaf.

Theorem 4.1 *The maximum path from the root to any leaf consists of at most* $\log_d N + 2$ *nodes and visits at most* $\log_d n + 1$ *different blocks.*

Proof The number of nodes on this particular path is obviously limited by the height of the heap. For $d \le B$ the root-block will contain the first $\lfloor \log_d B \rfloor$ levels of the heap, accordingly the first $\lfloor \log_d B \rfloor$ nodes on a path will be in that block. The remaining nodes will mostly be in different blocks since the steps between the nodes become bigger and bigger the further down the heap one comes. Therefore, the number of different blocks are $\log_d N + 1 - \lfloor \log_d B \rfloor \le \log_d \frac{N}{B} + 2 = log_d n + 2$. For $d > B$, the height is, as shown above, also $\log_d n + 1$. Which gives an upper bound on the number of different blocks visited. □

4.1.2 Insertion

An insert operation is made by increasing N by one and thereafter placing the new record in the right-most node (location $N-1$) and finally swapping upwards until the heap condition is fulfilled. The process of swapping a record upwards is also called *shift-up*.

Theorem 4.2 *One shift-up operation beginning at level l consists of at most l comparisons.*

Proof The swapping process continues until the root is reached, which gives an upper bound on the number of rounds of l, since l is the distance to the root, in each round two key values are compared, giving the claimed bound. □

Theorem 4.3 *One insertion has the internal complexity of* $O(\log_d N)$ *and the external complexity of* $O(\log_d n)$.

Proof An insertion always begins at level h or $h-1$ which means that the number of comparisons is at most $\log_d N$. From this it is clear that the internal complexity is $O(\log_d N)$. From Theorem 4.1 we can conclude that during one insertion $O(\log_d n)$ different blocks are visited, which is equal to the external complexity. □

From this observation it is obvious that the bigger fanout the better. The optimal situation is to have just the root and only leaves, which is the same as having $d = \theta(N)$. But this structure is not efficient with respect to delete-max, as will be shown next.

4.1.3 Delete-max

A record with a maximum key can be found and removed in constant time, but this can be done only once. To be able to use the heap for more than one delete-max, the heap has to be rebuilt so that the heap is complete and that the heap condition is fulfilled. This is done by moving the right-most record into the root, swapping this record with the biggest child record, until the heap condition is fulfilled. This process is named to shift-down.

Theorem 4.4 *One shift-down operation beginning at level l consists of at most $(d-1)(h-l)$ comparisons.*

Proof The number of rounds is at most $(h-l)$. To find the maximum among the d child records, requires $d-1$ key comparisons □

Theorem 4.5 *The internal complexity of one delete-max is $O(d \log_d N)$. The external complexity is $O(\frac{d}{B} \log_d n)$.*

Proof Since the shift-down of a delete-max always begins at level 1, the internal complexity must be $O(d \log_d N)$. There are d different alternatives to choose a maximum among the children from, the child records placed continuously in at most $\lceil \frac{d}{B} \rceil + 1$ blocks. Each round reads these $\lceil \frac{d}{B} \rceil + 1$ blocks. Giving an I/O-complexity of $O(\lceil \frac{d}{B} \rceil h)$, where h denotes the height of the heap. When $d \leq B$ the height is $\log_d N$, but as the top of the heap is kept in one block the I/O-complexity is: $O(\lceil \frac{d}{B} \rceil\ (\log_d N - \log_d B)) = O(\frac{d}{B} \log_d n)$. If $d > B$ then the height is $\theta(log_d n)$, and we can conclude that the I/O-complexity is $O(\frac{d}{B} \log_d n)$. □

The optimal d value depends on the number of insertions and deletions, we now assume an equal number of insertions and deletions. The internal complexity is minimized at $d = 3$ as shown in Appendix A (Proposition A.1). The external complexity is optimized at $d = B$, as shown in Appendix A (Proposition A.2).

4.1.4 Increase-key

This operation requires a separate method to retrieve the positions of records in the heap, to be able to adjust the key values of the records. Therefore we assume that an array is available that saves the positions of the records inserted. The actual increase is implemented by changing the key value of a record and thereafter shifting the record up. The position array must now be updated every time a pair of records in the heap are swapped (during shift-up and shift-down). This does not change the internal complexity of the operations described earlier. On the other hand, the external complexity deteriorates since the updates of the positions are not grouped in blocks, but are scattered arbitrarily among the position array.

Theorem 4.6 *When supporting increase-key the external complexity of insert is $O(\log_d N)$, and the external complexity of delete-max is $O(\frac{d}{B} \log_d n + \log_d N)$.*

Proof As mentioned the number of swaps and updates of the position array are the same. Since at worst each update causes an I/O, the external complexity is equal to the height of the d-heap, therefore $O(\log_d N)$. In the case of delete-max all the d children are accessed in every round causing $\lceil \frac{d}{B} \rceil$ I/O's therefore the I/O-complexity of delete-max is $O(\frac{d}{B} \log_d n + \log_d N)$. □

Theorem 4.7 *The internal and external complexities of increase-key are $O(\log_d N)$.*

Proof The complexity of increase-key depends on the complexity of shift-up, and therefore the internal complexity is $O(\log_d N)$. As with insertion each swap causes two updates therfore the external complexity is $O(\log_d N)$. □

4.2 B-trees as priority queues

In Section 3.2 we described how B-trees are used as dictionaries. In this section, we will describe how B-trees can be used as priority queues. The basic data structure is not changed, so we use some of the earlier algorithms as substitution for implementing the priority queue operations. Since we only wish to delete the maximum records, we introduce an internal buffer of size $B+1$. This buffer will contain an AVL-tree containing the records with largest key-values.

4.2.1 Insertion

Before a record is inserted in the structure, we check to see if it should be stored in the internal buffer. If not, the record is inserted as described in Section 3.2.4. If the record should be stored in the buffer, the record with the smallest key is removed from the buffer and inserted in the B-tree. The new record is inserted in the buffer.

Theorem 4.8 *The amortized internal complexity of insert is $\theta(\log_2 N)$.*

Proof As shown in Section 3.2.4, a normal insert has an internal complexity of $\theta(\log_2 N)$. Inserting in the buffer has an internal complexity of $O(\log_2 B)$, leading us to a total of $\theta(\log_2 N)$ (we assume that N is larger than B). $\square$

Theorem 4.9 *The amortized external complexity of insert is $\theta(\log_B n)$.*

Proof Follows from the fact that the external complexity of inserting a record in a B-tree is $\theta(\log_B n)$. $\square$

4.2.2 Delete-max

Delete-max simply deletes a record from the internal buffer. If the buffer is empty, we move the records from the rightmost leaf to the buffer. The parent of this leaf now contains one extra record and we move that record to the buffer as well. The parent node can become too small, so we perform CheckSize (see Section 3.2.5) on the parent node.

Theorem 4.10 *The amortized internal complexity of delete-max, is $O(\log_2 N)$.*

Proof Deleting a record from the buffer has an internal complexity of $O(\log_2 B)$. After refilling the internal buffer, we can carry out at least $\frac{B}{4} = O(B)$ delete-max operations before a new refill. Moving the AVL-tree from a leaf to the buffer and inserting a record from the parent, has an internal complexity of $\theta(B + \log_2 B)$. The CheckSize procedure has the internal complexity of $O((\log_2 B + B)(\log_B n + 1)) = O(\log_2 n + \log_2 B + B \log_B n + B) = O(log_2 N + B \log_B N)$. In all we get an internal complexity of $O(\log_2 B + \frac{B+\log_2 B+\log_2 N+B\log_B N}{B}) = O(\log_2 N)$. $\square$

Theorem 4.11 *The amortized external complexity of delete-max, is $O(\frac{1}{B}\log_B n)$.*

Proof Since CheckSize has an external complexity of $O(\log_B n)$ and is only performed every $O(B)$ times, the external complexity becomes $O(\frac{1}{B}\log_B n)$. $\square$

4.2.3 Increase-key

Since the B-tree is structured after the keys (priorities), rather than the record names, we need a second B-tree, where the records are structured after the record names. First, we locate the record in the record-name B-tree. Second, we use the records priority to delete the record from the priority B-tree. Third, we opdate the priority and insert the record into the priority B-tree again. This means that an increase-key operation consists of one retrieve, one delete, and one insert operation.

Theorem 4.12 *The amortized internal complexities of the insert, delete-max and increase-key operations are $\theta(\log_2 N)$ and the amortized external complexities are $\theta(\log_B n)$, when supporting increase-key .*

Proof Since the insert and increase-key operations only use standard B-tree operations, the bounds follow from section 3.6. Unfortunately the use of two B-trees, in which one of them has not an internal buffer, makes the complexities of delete-max worse. Since a delete-max operation on the priority B-tree is followed by a delete on the record-name B-tree, the external complexity of a delete-max operation becomes $\theta(\log_B n)$ (follows from Section 3.6). The internal complexity does not change. This means that the use of a buffer has no effect when we intend to use increase-key in the operation repertoire. □

4.3 I/O-queues

In the following we will present a priority queue that is designed to work efficiently in external memory. The main idea is to buffer records in internal memory and only access the external memory when one of the buffers in the internal memory is filled up. Since the structure is I/O-optimal, in the sense defined in the chapter introduction, we name it the *I/O-queue*. An article version is found in [**?**].

4.3.1 The data structure

We define a tree structure with nodes of size M. One node can contain M records and is eventually parent for up to m children. A node which is a parent is an *internal node* otherwise it is a *leaf node*. The nodes are placed so that the tree is kept *complete*, i.e. all internal nodes, except the rightmost internal node, must have m non-empty children. The *rightmost* node is a special node, always pointed out during execution, intuitively it is the rightmost leaf when illustrating the tree. The *rigthmost internal node* (which is referred to above) is the parent of the rightmost leaf. The heap condition states that any key in a parent is greater than or equal to any keys stored in the children, which means that the root of the tree contains the greatest of the keys. Each node stores the records in a sorted order with the records filling the node from beginning to end. Further, each node stores the sizes of the children together with its own records. This means that when the size of a node is needed, one has to consult the parent node.

One size index points out a position in the range of $\{0..M-1\}$. All in all the number of bits used for storing the pointers will be $m\lceil\log_2 M\rceil$ per node. For the case of simplicity we assume that each node can contain the M records, plus the required indices. In practice the solution is of course not so simple.

Besides the usual heap characteristics we define the following invariant on all the nodes, except for the root and the rightmost node: The nodes should contain a certain fraction of M records. For simplicity we choose that all nodes should always have at least $M/2$ records. The number of B-blocks containing records per node is between m and $\lceil m/2\rceil$. The invariant ensures that the number of B-blocks occupied by the tree is proportional, within a factor 2, to the number of records currently in the structure:

Records in the structure N

Size of the tree in nodes $s \leq N/\lceil \frac{M}{2} \rceil \leq \frac{2N}{M}$

Number of B-blocks $n = sm \leq \frac{2N}{M}\lceil \frac{M}{B} \rceil \leq \frac{2N}{M}(\frac{M}{B}+1) = \frac{2N}{B} + \frac{2N}{M} \leq \frac{4N}{B}$

Height of structure $h = \theta(\lceil \log_m s \rceil + 1) = \theta(\log_m \frac{n}{m} + 1) = \theta(\log_m \frac{N}{M} + 1)$

Since the tree is complete and the nodes are placed in succession, the location of the nodes can be calculated like with d-heaps (Section 4.1). The locations are numbers in the range $[0 \ldots R]$ where R is the location of the rightmost leaf, initially $R = 0 =$ the root.
$parent(x) = \lfloor \frac{x-1}{m} \rfloor$, for $x \in \{1 \ldots R\}$
$children(x) \in \{mx+1, \ldots, m(x+1)\} \cap \{0, \ldots, R\}$

A priority queue should support insert and delete-max operations for single records. We will buffer both kind of operations by buffering the records involved in the operations in internal memory. This buffering is to avoid I/O-costs every time one of the operations is executed. The root of the structure is kept resident to facilitate delete-max and the most recent inserted records are kept separate from the rest of the structure to facilitate fast insertions. The internal memory is divided in three parts:

OUT Which is the root of the structure containing the records with the greatest key-values.

IN Which contains the most recently inserted records.

ADM Which stores the $\theta(m)$ indeces to the children of the root.

ADM uses $\theta(m \log_2 M)$ bits, but as mentioned we assume that the m-blocks can contain these bits besides the M records. The M records are divided so that OUT contains cM records and IN has $(1-c)M$ records for some $0 \leq c \leq 1$.

4.3.2 Insertion

Before the records in IN are inserted into the main structure, some of the records might be removed (during delete-max). To ease the removal these records are kept as a 2-heap. The insertion of the IN records into external memory works by first sorting the records (for example by internal heapsort since the heap is built). Afterwards as many records as possible are moved to the rightmost leaf, if the rightmost leaf is full, a new node is allocated and made rightmost leaf, the new rightmost leaf is filled with remaining records. As mentioned, the nodes are placed in a row, therefore the new node is placed next to the old rightmost leaf in external memory. Finally the new records are moved upwards to fulfill the heap condition.

```
Insert(r: record, P: I/O-queue)
  If IN full Then Flush()
  2-heap-insert(r, IN)

Flush()
  Heapsort(IN)
  While rightmost leaf NOT full And still records in IN
    move record from IN to rightmost leaf
  Shift-up(rightmost leaf)
```

```
  If still records in IN
    Allocate node
    rightmost block=new node;
    While still records in IN
      move record from IN to rightmost leaf
    Shift-up(rightmost leaf)

Shift-up(b: m-block)
  Fulfill-heap-condition(b, parent(b))
  If b changed and parent(b)>0
    Shift-up(parent(b))

Fulfill-heap-condition(b, p: m-block)
  temp=2-way-Merge(b ,p)
  b=first size(b) records of temp     { size(x) is the number of records in   }
  p=last size(p) records of temp      { m-block x, not the number of B-blocks }
```

The movement of a record from IN to a rightmost leaf is made blockwise, so that the records are clustered in a B-block and moved by a block write operation. The fulfillment of the heap condition between a node and its parent is carried out by merging the records of the two nodes records into a workspace, and then by moving the records back to the two nodes so that the biggest records are placed in the parent node. To be sure of keeping the heap condition valid between the parent and any other children nodes, there cannot be moved more records to the parent than there were originally. The remaining records are returned to the node. See Figure 4.1.

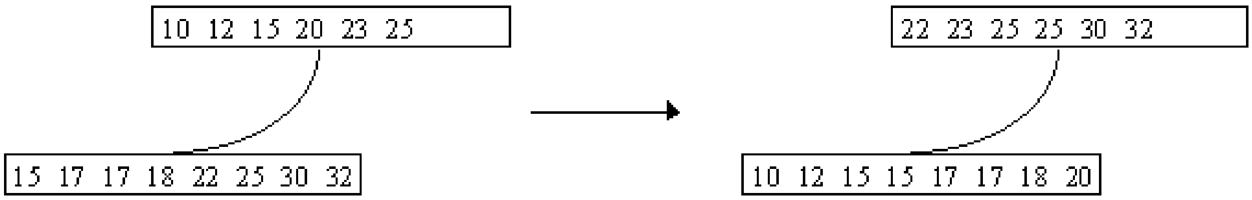

Figure 4.1: Before and after shift-up

Theorem 4.13 *One shift-up on a structure with N records has the I/O-complexity of $O(m \log_m n)$ and the internal complexity of $O(M \log_m n)$.*

Proof At worst a shift-up begins at the bottom level and continues all the way to the root, i.e. the number of rounds is $O(\log_m n)$. Each round involves the 2-way merging and movement of at most $2M$ records. As shown in Section 2.4.3 2-way merging is made with $O(m)$ I/O's and $O(M)$ internal work. Therefore, one shift-up requires $O(m \log_m n)$ block transfers and $O(M \log_m n)$ internal operations. □
The total work done after every $O(M)$ insertions will be charged to each of the $O(M)$ operations. This gives:

Theorem 4.14 *One insertion has an amortized I/O-complexity of $O(\frac{1}{B} \log_m n)$ and an amortized internal complexity of $O(\log_2 N)$. where N denotes the current size of the structure in records.*

Proof The external work done is $\theta(m)$ when moving the records of IN, and that for the one or two shift-up operations, $O(m \log_m n)$. By dividing the work among the $O(M)$ operations we get the amortized I/O-complexity of $O(\frac{1}{B} \log_m n)$.

The internal complexity is a mixture of amortized and worst-case complexity. The insertion into the internal heap cost at worst $O(\log_2 M)$ per insert operation. The sorting of the internal heap costs $O(M \log_2 M)$ and the shift-ups costs $O(M \log_m n)$. The amortized cost of sorting and shifting up is $O(\log_2 M)$ and $O(\log_m n)$ respectively. Adding up the three factors gives:
$O(\log_2 M + \log_2 M + \log_m n) = O(\log_2 M + \log_2 \frac{N}{B}) = O(\log_2 N^2) = O(\log_2 N)$. □

4.3.3 Delete-max

When deleting the maximum record it is either in IN or in OUT. But if OUT is empty, the maximum record could be on the next level of the structure and we therefore have to fill up the root.

```
Delete-max(P: I/O-queue)
  If OUT is empty Then Refill(OUT)
  Remove and return the maximum of OUT and IN

Refill(b: m-block)
  If b is a leaf Then
    Leaf-melt(b)
  Else
    While children not empty and b not full
      m-way merge from the children of b into b
    2-way merge the new and eventually existing list of records in b
    For all less than half-full children, c, REFILL(c)

LEAF-MELT(b: m-block)
  If next-to-melt=-1 or next-to-melt=b Then  { next-to-melt is a global      }
    next-to-melt=b                           { pointer indicating an eventual }
  Else                                       { less than half full leaf      }
    b=2-way-merge(b, next-to-melt)
    Shift-up(b)
    move rightmost node to next-to-melt's position
    Shift-up(next-to-melt)
    rightmost=rightmost-1
    If b<M/2 Then next-to-melt=b
    Else next-to-melt=-1;
```

First we will describe the refill of the internal nodes and afterwards the melting of leaves. The tail-recursions in the refill procedure ensure that the children of the current node are kept at least half full after a refill. In the example of Figure 4.2 the node to be filled is less then half full. The needed selection pointers and temporary workspaces are not shown.

Theorem 4.15 *The filling of an internal node has I/O-complexity $\theta(m)$ and internal complexity $\theta(M \log_2 m)$.*

Proof A m-way merge of the internal nodes will move between $\frac{M}{2}$ and M records every time, since each child node contains at least $\frac{M}{2}$ records and since the node being filled can contain at most M records. In Section 2.4.4 m-way merging of $\theta(M)$ records is shown to have I/O-complexity $\theta(m)$ and internal complexity $\theta(M \log_2 m)$. Note that the tail-recursions at the end of the procedure do not affect the complexities of the merging. □

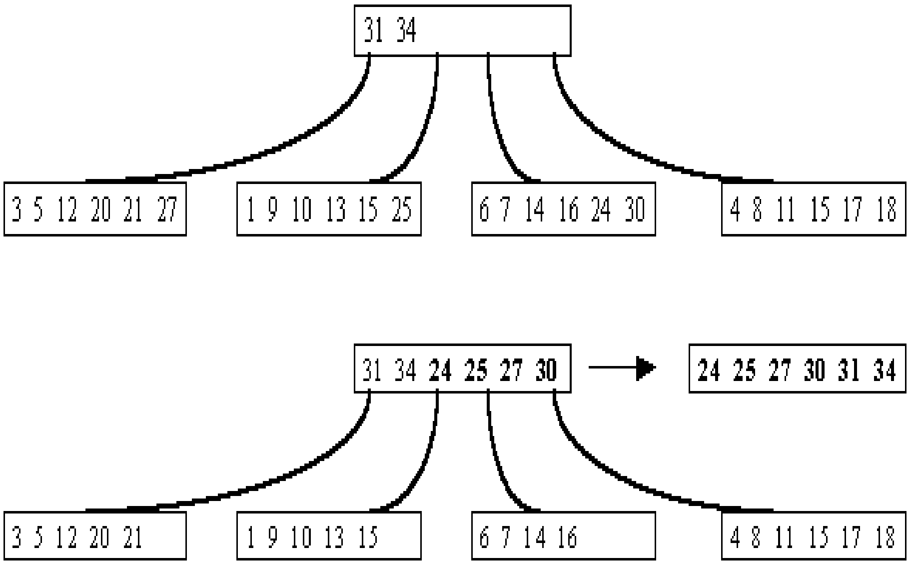

Figure 4.2: Illustration of a refill

The melting of two leaves is carried out to maintain the compactness and completeness of the structure such that the tree shrinks and keeps the proportion between the number of records and the size of the current structure. The structure will have at most one leaf which is less than half full at the time. This one leaf is melted with the next leaf that becomes less than half full, disregarding the rightmost leaf.

Theorem 4.16 *The melting of two leaf nodes has an I/O-complexity of $O(mlog_m n)$ and an internal complexity $O(Mlog_m n)$.*

Proof The actual movements and the 2-way merging of the records from one leaf block to another have the linear complexities $\theta(m)$ and $\theta(M)$ respectively. The expensive part of the melting are the two shift-up operations, which as shown in Theorem 4.13, have the postulated complexities. □

Theorem 4.17 *One delete-max has an amortized I/O-complexity of $O(\frac{1}{B}\log_m n)$ and an amortized internal complexity of $O(\log_2 N)$ N is the current size of the structure.*

Proof The complexities will be shown by an amortized analysis using the banker's view as described in [40]. Using this approach we attach credits to each record on insertion, and show that the summarized credits suffice to cover the total work.

First we show the external complexity. We state that each record as long as it is in external memory has a credit of $O(\frac{1}{B}level)$, where *level* is the level the record is at currently, counting from the root and down. All records are flushed to leaf level in the beginning. At that time each record has the initial credit of $O(\frac{1}{B}\log_m n)$, this credit is paid for by the insert operation (along with the other costs of inserting records). We will show that this credit is enough to let each record, via refills, reach the root.

We will distinguish between the three movement directions of the records: down, up and sideways.

Case down: A record moves down only if it is swapped during the shift-up of another record. In this case the cost of the movement is assumed to be paid for by the shift-up operation. But we also let the records swap their credits so that the credit still follows the level.

Case up: A record moves up either during refill or shift-up. The shift-up case happens during an insertion or during a leaf-melt. The leaf-melt case is discussed below, but we can assume that the leaf-melt covers for shift-up. The shift-up caused by the flush of records during insert is shown to be covered in Theorem 4.14. As shown above, the credits and level positions follow each other after a shift-up. The upwards movement during refill is where the credit is used. The $O(M)$ records which are moved during a refill at the time, split cost of $O(m)$ (see Theorem 4.15) evenly. Which means that one record pays $O(\frac{1}{B})$, but also moves one level up. Therefore the credit and level still go along.

Case sideways: We have now shown that a record has enough credit, given at insertion time, to reach the root. But we also have to meet the costs of the leaf-melts. The costs are covered by the records that are deleted. So that every record gives a 'payment' of $O(\frac{1}{B}\log_m n)$ to meet the next leaf-melt that will occur when it exits the structure. The two credits, insertion credit and delete credit, are not the same since N can be different at the two different times. We can be sure that exit credits are sufficient to meet the leaf-melt costs of $O(mlog_m n)$ (Theorem 4.16), since prior to each leaf-melt there must have been $O(M)$ deletes. The size of each credit is also enough since the height of the structure on exit time for the $O(M)$ records is the same as height on the time for the leaf-melt.

We can conclude that the insertion operation pays for insertion and movement of the records towards the root. While the delete-max operation pays for the contraction of the structure, caused by the removals.

The internal complexity is shown in a similar manner. Here the initial insertion credit is $O(\log_2 m \; level) = O(\log_2 m \log_m n) = O(\log_2 n)$ which is given when the record is inserted. Like before during shift-up, whether the record is moved up or down, credits are swapped but no credits are used to pay for the shift-up. The costs of the swapping rely on the operation that uses shift-up (Flush and Leaf-melt). Since a refill merging costs $O(M \log_2 m)$, each record pays $O(\log_2 m)$, as wanted to keep the relation between credit and level. The internal costs of $O(M \log_m n)$ when executing a leaf-melt requires an exit credit of just $O(\log_m n)$ per record. Besides this every delete-max performs an extraction of the internal 2-heap, which costs $O(\log_2 M)$. In all the internal cost is at most $O(\log_2 N)$ per record. $\square$

4.3.4 Change-key

If we wish to have a structure that supports changes of key-values while keeping the size of the structure proportional to the number of records inserted, we will need some way to locate the wanted records. Searching seems to be done only in linear time. Therefore it will require some auxiliary structure to keep track of the positions of the records. One way to implement this structure is an array, indexed by the id's of the records. We shall see that the I/O-complexity grows from $O(\frac{1}{B}\log_m n)$ to $O(\log_m n)$, where N is the number of records inserted. The increase in the complexity is caused by the updates of the position array dominating the actual work done. The complexity of keeping the array up to date, while performing the operations on the main structure, is proportional to the number of moves of the records within the main structure. The number of moves are not the same as the internal complexity, which also covers the number of comparisons of records.

The change of a key value is implemented as a delete operation and insert operation of the same record with the new key value. The deletion is made lazy as long as the node where the record is located has more than $M/2$ not deleted records. If the node has less than $\frac{M}{2}$ records, then the node is refilled and the deleted records are skipped. The insert and delete-max operations are implemented in the same way as above with fixed key-values. The difference is the updating of the position array and the possibility of deleted records within the blocks, also named *holes*, they are identified for example with the key value $-\infty$. When encountering holes, during the different situations of merging, they are skipped by simply jumping to the next record.

```
Change-key(r-id: record-id, k: value, P: I/O-queue)
  pos=Position(r-id)
  r=Delete(pos)
  r.key-value = k
  Insert(r)

Delete(pos: position)
  If pos in IN Then make 2-heap-delete(pos, IN)
  Else
    load B-block number pos/m+pos%m
    delete record at position pos in loaded B-block
    write back B-block
    count down node-locations size (located at the parent)
    If b is less than half-full then
      remove holes in b by contracting the records
      REFILL(b)
  return the deleted record
```

I/O-complexity

As shown in the application with the fixed key-values, insertion and delete-max is met by a credit of $O(\frac{1}{B}\log_m n)$, but because of the updates of the position array the credit has to be $O(\log_m n)$ per operation. Which we will show below, covers the I/O costs.

Theorem 4.18 *One shift-up has the amortized external complexity of* $O(\log_m n)$

Proof The external complexity of the 2-way merge is now $O(M)$ because the relocation of the $O(M)$ records is updated using $O(M)$ I/O's. One whole shift-up now has the external complexity of $O(M \log_m n)$ or the amortized complexity $O(\log_m n)$. □

In the same way one refill is shown to cost $O(M)$, since we move $O(M)$ records.

Theorem 4.19 *The external complexity of the insert, delete-max and change-key operations in a structure supporting change-key is* $O(\log_m n)$ *where* N *is the number of records in structure*

Proof The insert and delete-max operations are easily shown to be $O(\log_m n)$ since the cost of shift-up is $O(\log_m n)$ per record and the cost of refilling a node is (M) or $O(1)$ I/O's per node per level. Deleting a record, during change-key, within an arbitrary node has the I/O-complexity of $O(1)$, because of the constant number of block reads. The created holes from $O(M)$ deletes causes $O(M)$ skips which gives $O(m)$ I/O's, and therefore $O(\frac{1}{B})$ I/O-cost on each change-key operation. If the deletion causes a refill then previously $O(M)$ delete-max or change-key operations have occurred, these operations together meet the cost refilling. The only costs left to add are the insertion costs, the cost of change-key is therefore $O(1 + \frac{1}{B} + \log_m n) = O(\log_m n)$. In these proof we have assumed that $O(\log_m n + 1) = O(\log_m n)$ because the $\log_m n =$ the height, is at least one. □

Internal complexity

The updates of the position array do not increase the internal complexity of the operations, since one update is simply one write. The created holes do not change the already shown bound of $O(\log_2 N)$, as we shall see. As in the above section the $O(M)$ extra readings caused by $O(M)$ holes are easily met by the $O(M)$ records which have their values changed. If $O(M)$ deletes cause a refill of $O(M)$, these are also met.

4.3.5 Increase-key

If we allow that the complexities are proportional to the number of operations, and only allow upgrading of priorities (contrary to the general change of key-values) we can show an implementation that has the external complexity $O(\frac{1}{B}\log_m n+1)$, where N is the number of operations. This is done by simply inserting the upgraded record as a new record and keeping the old copy of the record in the structure (somewhere unknown). This requires that the record is a parameter to the increase operation.

First we notice that since we only upgrade priorities (which in this case means increasing key-values), the latest inserted instance of a record and the instance with the highest priority is the same record. After the record has been removed as a maximum record we assume that it will not be re-introduced to the structure. Any remaining occurrences of the record are to be considered as garbage. We will keep one bit of information for each record to be inserted, which tells whether it has been removed.

```
Increase-key(r: record, P: I/O-queue)
  Insert(r, P);
```

Note that the increase-key operation demands a whole record instead of just the record id. The complexities are easily established since we use only the operations described earlier, with the extension that the height of structure is dependent on the number of insertions and not on the number of currently inserted records. The only difference is the refilling of the root. Every time the root is filled, the records are scanned and outdated records are removed. If thereby the root is emptied, it is refilled once more.

The extra work of updating when deleting the maximum record gives at worst an extra I/O of $O(1)$ per operation. The checking of the records in the root gives an extra work of $O(1)$ per insertion (because the insertions meet the costs of the refills). The external complexity of all three operations is now $O(\frac{1}{B}\log_m n+1) = O(\frac{1}{B}(\log_m n + B))$

4.4 Buffer trees as priority queues

The buffer tree structure described in Section 3.3 can also be used for priority queuing. For a detailed description see [6].

The delete-max operation is implemented by pushing all buffers along the rightmost path, so that the maximum records are placed in the rightmost leaf blocks. The contents of these blocks are then moved to a buffer in internal memory which therefore contains $O(M)$ records. Pushing the buffers is made $\log_m n$ times, and every push costs $O(m)$. Since the operation only occurs every $O(M)$ delete-max, the amortized external complexity per delete-max is $O(\frac{1}{B}\log_m n)$. When executing a delete-max, the maximum record from this buffer and operation buffer is extracted. The internal complexity is $O(\log_2 N)$.

Insertions are buffered and pushed downwards when the buffers overflow (as described in Section 3.3). Since the number of insert operations and the number of records inserted are the same, and since delete-max does not “insert” a new record (as the delete operation does in Section 3.3), the size of the queue only depends on the number of inserted records. This means that the external complexity of inserting a record now is $O(\frac{1}{B}\log_m n)$ and the internal complexity is $O(\log_2 N)$.

4.5 Conclusions

To conclude on the structures that only support insert and delete-max, we first list the complexities of those in Table 4.1. If we choose $d = B$ for the d-heap (i.e. a B-heap),

		d-heap	B-trees	I/O-queue	Buffer tree
Insert	internal	$O(\log_d N)$	$\theta(\log_2 n)$	$O(\log_2 N)$	$O(\log_2 N)$
	external	$O(\log_d n)$	$\theta(\log_B n)$	$O(\frac{1}{B}\log_m n)$	$O(\frac{1}{B}\log_m n)$
Delete-max	internal	$O(d\log_d N)$	$O(\log_2 n)$	$O(\log_2 N)$	$O(\log_2 N)$
	external	$O(\frac{d}{B}\log_d n)$	$O(\frac{1}{B}\log_B n)$	$O(\frac{1}{B}\log_m n)$	$O(\frac{1}{B}\log_m n)$

Table 4.1: Complexities of various priority queue structures not supporting increase-key.

the B-heap and the B-tree have the same insert I/O-complexity. But an insertion in a B-tree always means a traversal from the root to a leaf. In B-heaps this is not always the case. This means that B-heaps are cheaper during insertion. On the other hand delete-max operations are cheaper using B-trees compared to B-heaps, cf. the complexities $O(\frac{1}{B}\log_m n)$ and $\theta(\log_B n)$.

The external complexities for the I/O-queue and the Buffer tree does not indicate which of the two is the better. It seems, though, that the I/O-queue is simpler, because inserting $O(M)$ records involves less nodes. The I/O-queue uses 2-way merging and the Buffer tree uses m-way partitioning, this means that even though the I/O-complexity is the same more B-blocks are touched in the Buffer tree when inserting. After every $O(M)$ delete-max operations the same number of nodes manipulated on the average in the two structures.

The structures in Table 4.2 also support increase-key. N_O is the number of operations

		d-heap	B-trees	I/O-queue w. change-key	I/O-queue w. increase-key
Insert	internal	$O(\log_d N)$	$\theta(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N_O)$
	external	$O(\log_d N)$	$\theta(\log_B n)$	$O(\log_m n)$	$O(\frac{1}{B}\log_m n_O + 1)$
Delete-max	internal	$O(d\log_d N)$	$\theta(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N_O)$
	external	$O(\frac{d}{B}\log_d n + \log_d N)$	$\theta(\log_B n)$	$O(\log_m n)$	$O(\frac{1}{B}\log_m n_O + 1)$
Increase-key	internal	$O(\log_d N)$	$\theta(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N_O)$
	external	$O(\log_d N)$	$\theta(\log_B n)$	$O(\log_m n)$	$O(\frac{1}{B}\log_m n_O + 1)$

Table 4.2: Complexities of various priority queue structures supporting increase-key.

and $n_O = \theta(\frac{N_O}{B})$.

B-trees are the only structure that is dynamic, the others require that the number of different records to be inserted, is known in advance. In many applications this is known, therefore one of the two I/O-queues is preferable. To decide between the two we compare the complexities:

$$\log_m n < \frac{1}{B}\log_m n_O + 1$$
$$\Rightarrow \log_m n < log_m(m \sqrt[B]{n_O})$$
$$\Rightarrow n < m \sqrt[B]{n_O}$$
$$\Rightarrow n_O > (\frac{n}{m})^B$$

Since $\frac{n}{m} > 1$ the I/O-queue with increase-key is the realistic choice for every practical application.

Apparently it is not possible to implement a structure supporting increase-key that has an amortized I/O-complexity of $O(\frac{1}{B}\log_m n)$. We postulate that the lower limit of increase-key is $\Omega(max(1, I/O_{work})) = \Omega(I/O_{work} + 1) = \Omega(I/O_{work} + \frac{B}{B})$, where I/O_{work} is the actual work done on the priority queue when increasing. Our argumentation rests on two assumptions concerning the increase-key operation:

- Each record is equally probable to be increased.
- The increased record has to be done real-time. I.e. the priorities have to be changed within a constant time, to ensure the integrity of the queue.

Since all records are equally likely to be increased, we cannot buffer and not in general foresee the pattern of the increasing.

Chapter 5

Sorting

In this chapter we will discuss both the internal and external complexity of sorting. The external sorting problem is defined as follows: Initially, the internal memory is empty, and the N records to be sorted are residing continuously in external memory. After the sorting, the disk contains the N records, sorted in a non-decreasing order. We do not demand that the sorted records are residing in the same place as initially. The internal sorting is defined likewise, we just assume some sort of virtual memory management. In both cases we are sorting arrays.

In [4] Aggerwal and Vitter show that the lower bound, in terms of external complexity, of permuting N records is

$$\Omega(min\{N, n\frac{\log_2(1+n)}{\log_2(1+m)}\}).$$

Since sorting is a special case of permuting, these lower bounds are valid for sorting. When $B\log(m) = o(\log(n))$, or B and M are extremely small with respect to N, we get a lower bound of $\Omega(N)$, which is not very good. Aggerwal and Vitter shows that in this case we can prove a better lower bound, so that the lower bound for sorting N records, is

$$\Omega(n\frac{\log_2(1+n)}{\log_2(1+m)}) = \Omega(n\log_m n).$$

It is interesting to note that the hard part of sorting is moving the records to the right places, not determining the record-order [44]. This means that even if we knew where to put each record, the external complexity would not decrease.

5.1 Mergesort

The basic idea of mergesort is to break the input into smaller pieces, sort each of the pieces separately, and merge them.

5.1.1 Internal mergesort

The internal mergesort splits the input into two pieces and sorts each of them recursively, after which the pieces are merged together. By internal we mean that the algorithms assume some virtual memory management. The algorithm is shown below:

```
Mergesort(A : Array, begin : index, end : index)
   if end > begin
      Mergesort(A,begin,(end-begin)/2)
      Mergesort(A,(end-begin)/2+1,end)
      2-way-merge(A[begin..(end-begin)/2], A[(end-begin)/2+1..end])
```

Theorem 5.1 *The internal complexity of the internal mergesort is $\theta(N \log_2 N)$.*

Proof The internal complexity can be described by the recurrence $IC(N) = \theta(N) + 2IC(\frac{N}{2})$, which by the Master Theorem of [16] gives $IC(N) = \theta(N \log_2 N)$. □

Theorem 5.2 *The external complexity of the internal mergesort is $\theta(n \log_2 \frac{n}{m})$.*

Proof The external complexity can be described by the recurrence $EC(N) = \theta(n) + 2EC(\frac{N}{2})$, which by the Master Theorem of [16], gives $EC(N) = \theta(n \log_2 N)$. When end-begin is smaller than M, the recurrence does not result in reading any new blocks from external to internal memory. This means that the external complexity is $\theta(n \log_2 N - n \log_2 M) = \theta(n \log_2 \frac{n}{m})$. □

5.1.2 External mergesort

In this section we will show that a m-way mergesort is I/O-optimal. We start by sorting each m-block and then merge these m-blocks m-way, recursively. In the algorithm shown below we assume that $\frac{N}{B}$ is a whole number.

```
ExtMergesort(A : Array, BeginBlock : index, EndBlock : Index)
   sort all m-blocks of a using internal Mergesort
   MakeMerge(A, BeginBlock, EndBlock)

MakeMerge(A : Array, BeginBlock : index, EndBlock : index)
   if EndBlock - BeginBlock > m
      PieceSize = (EndBlock - BeginBlock)/ m
      for t = 1 to m
         MakeMerge(A, BeginBlock + (t-1)*PieceSize, BeginBlock + t*PieceSize)
      merge the pieces m-way
```

Theorem 5.3 *The internal complexity of external mergesort is $\theta(N \log_2 N)$.*

Proof The initial sorting of all the m-blocks has an internal complexity of $\theta(\frac{n}{m} M \log_2 M) = \theta(N \log_2 M)$ (since $\frac{n}{m} = \frac{N}{M}$). In the function MakeMerge we first call recursively m times. The m-way merge of N records has an internal complexity of $\theta(N \log_2 m)$ (see Section 2.4.4). The MakeMerge then has an internal complexity of $IC_{MakeMerge}(N) = \theta(N \log_2 m) + mIC_{MakeMerge}(\frac{N}{m})$, which, by the Master Theorem is $\theta(N \log_2 m \log_m N) = \theta(N \log_2 m \frac{(\log_2 N)}{(\log_2 m)}) = \theta(N \log_2 N)$. Since the recursion stops when $EndBlock - BeginBlock > m$, the correct complexity for MakeMerge should be $\theta(N \log_2 N - N \log_2 M) = \theta(N \log_2 \frac{N}{M})$ This means that the total complexity is $\theta(N \log_2 M + N \log_2 \frac{N}{M}) = \theta(N \log_2 N)$. □

Theorem 5.4 *The external complexity of external mergesort is $\theta(n \log_m n)$.*

Proof The initial sorting has an external complexity of $\theta(n)$. To merge the pieces m-way (in the function MakeMerge) we only need one block of each piece to be merged into the internal memory, that is, we can make a m-way merge using $\theta(n)$ I/O's while merging N records. This gives us the recurrence $EC_{MakeMerge}(n) = \theta(n) + mEC_{MakeMerge}(\frac{n}{m})$, which gives the optimal external complexity of $\theta(n \log_m n)$. □

5.2 Samplesort

The basic idea of samplesort is to select a number of sample keys from the input array randomly, sort the samples and use them as pivot elements in a splitting of the input. The sequences produced by the splitting are then sorted recursively and the sequences are put together. We will in the following find a limit to the recursion depth which is used to establish the internal and external complexities of the samplesort algorithms.

A sequence in every recursion is split in k subsequences (using $k-1$ pivot elements).

Theorem 5.5 *The probability that none of the k subsequences is longer than $\frac{N}{k}\log_2 N^c$, where c is a positive constant, is $1-\frac{1}{N^c}$*

Proof Prob(all subsequences $\leq \frac{N}{k}\log_2 N^c$)

$= 1-$Prob(one subsequence $> \frac{N}{k}\log_2 N^c$)

$= 1-$Prob(at least $\frac{N}{k}\log_2 N^c$ records have not been selected as pivot elements)

$= 1-(1-\frac{k}{N})^{\frac{N}{k}\log_2 N^c}$, since the pivots elements are selected randomly

$= 1-((1-\frac{k}{N})^{\frac{N}{k}})^{\log_2 N^c}$

$= 1-(e^{-1})^{\log_2 N^c}$, for large N's

$> 1-e^{-\log_2 N^c}$

$> 1-\frac{1}{N^c}$ □

When considering the internal complexity the recursion ends when a sequence has a length less than or equal one. The maximum recursion depth we denote the *internal recursion depth.*

Theorem 5.6 *Then internal recursion depth is $O(\log_k N)$.*

Proof With sequence size N the maximum size of each subsequence is $\frac{N}{k}\log_2 N^c$ (with high probability). We assume that $\frac{N}{k}\log_2 N^c < N$, otherwise the recursion would never stop. This leads to a limit on c of $\frac{k}{\log_2 N}$. In the first recursion the maximum size of the sequences are $\frac{N}{k}\log_2 N^c$. In the second recursion the maximum sizes are:

$\frac{\frac{N}{k}\log_2 N^c}{k}\log_2(\frac{N}{k}\log_2 N^c)^c$

$< \frac{N}{k^2}\log_2 N^{2c}$, since $\frac{N}{k}\log_2 N^c < N$

This leads to the maximum size at the ith recursion being $\frac{N}{k^i}\log_2 N^{ic}$. The maximum recursion depth h is found by the following equation:

$\frac{N}{k^h}\log_2 N^{hc} \leq 1$

$\Rightarrow \frac{k^h}{h} \geq cN\log_2 N$

$\Rightarrow h\log_2 k - \log_2 h \geq \log_2(cN\log_2 N)$

$\Rightarrow h\log_2 k \geq \log_2(cN\log_2 N)$

$\Rightarrow h \geq \log_k c + \log_k N + \log_k \log_2 N$

Since $c < N$ for $k < N\log_2 N$ and $k \leq N$ we conclude that h is $O(\log_k N)$. □

When considering the external complexity the sequence sizes are measured in blocks. The recursion does not cause any block transfers when a sequence is smaller then m blocks. The maximum recursion depth we denote *external recursion depth.*

Theorem 5.7 *The external recursion depth is $O(\log_k \frac{n}{m})$.*

Proof The proof is similar to the proof of the internal recursion depth. The maximum size of the ith recursion is $\frac{n}{k^i}\log_2 n^{ic}$. This means that the maximum recursion depth h fulfils the following inequality:

$$\frac{n}{k^h}\log_2 n^{hc} \leq m$$

$$\Rightarrow h \geq \log_k c + \log_k \frac{n}{m} + \log_k \log_2 n$$

This leads to $h = O(\log_k \frac{n}{m})$. □

5.2.1 Internal samplesort

The internal samplesort (also known as quicksort) is a samplesort that selects only one pivot element. The algorithm is shown below:

```
Samplesort(A : Array, begin : index, end : index)
   if end > begin
      select one pivot element p from A randomly
      split A in two sequences according to p
      Samplesort(A, begin, FirstSequenceEnd)
      Samplesort(A, FirstSequenceEnd + 1, end)
```

Theorem 5.8 *The internal complexity of the internal samplesort is $O(N\log_2 N)$.*

Proof At each of the recursion levels each of the N records are compared and moved a constant number of times. By using Theorem 5.6 with $k = 2$ we get the internal recursion depth of $O(\log_2 N)$, leading us to an internal complexity of $O(N\log_2 N)$. □

Theorem 5.9 *The external complexity of the internal samplesort is $O(n\log_2 \frac{n}{m})$.*

Proof On each of the recursion levels each of the $O(n)$ blocks is read once and written once, since the sequences are read by scanning. We assume that the head blocks of the sequences used during a split are kept in internal memory. By using Theorem 5.7 with $k = 2$ we get the external recursion depth of $O(\log_2 \frac{n}{m})$, giving us an external complexity of $O(n\log_2 \frac{n}{m})$. □

5.2.2 External samplesort

The external samplesort, selects $m - 1$ pivot elements randomly and sort these. Then we scan the input from the beginning, moving each record to a new sequence, according to its value. $Sequence_i$ contains all the records with value between $pivot_{i-1}$ and $pivot_i$. The algorithm is shown below:

```
SampleSort(A: array, begin,end: index)
   if end > begin
      k = min(m,end-begin+1)
      Select k-1 pivot elements randomly
      Sort the pivot elements
      for t = begin to end
        move record A[t] to the correct sequence in temporary workspace
      move sequences to A
      for t = 1 to k do
         SampleSort(A, BeginSequence(t), EndSequence(t))
```

The algorithm assumes that the internal memory has size $m + 1 + o(m)$ blocks, m for the output buffers, one for the input buffer and $o(m)$ for the pivot elements. In practice the number of output buffers will be smaller so that the blocks in use are at most m.

Theorem 5.10 *The internal complexity of external samplesort is $O(N \log_2 N)$.*

Proof To begin with, we will not consider the internal complexity of selecting the m-1 pivot elements and sorting them. We can select which sequence to move a record to in time $O(\log_2 m)$, by a binary search of the sorted pivot elements. This means that moving all records on a recursion level to the appropriate sequence costs $O(N \log_2 m)$. By using Theorem 5.6 with $k = m$ we get the internal recursion depth of $O(\log_m N)$. The total complexity of splitting is $O(N \log_2 m \ \log_m N) = O(N \log_2 N)$. The internal complexity of selecting the m-1 pivot elements and sorting them is $O(m + m \log_2 m) = O(m \log_2 m)$. This is done one time at the first recursion, m times at the second recursion, m^2 time at the third, and so on until the second last recursion is reached (at the last recursion there is only one element). This means that the internal complexity can be described as:

$O(m \log_2 m \ \sum_{i=0}^{h} m^i), h = \log_m N - 1$

$= O(m \log_2 m \ \frac{m^{h+1} - 1}{m-1})$ (Appendix A Proposition A.3)

$= O(m \log_2 m \ \frac{m^{\log_m N - 1 + 1} - 1}{m-1})$

$= O(m \log_2 m \ \frac{N}{m-1})$

$= O(N \log_2 m)$

The total internal complexity becomes $O(N \log_2 N + N \log_2 m) = O(N \log_2 N)$. □

Theorem 5.11 *The external complexity of external samplesort is $O(n \log_m n)$.*

Proof The external recursion depth is $O(\log_m \frac{n}{m}) = O(\log_m n - 1)$ (Theorem 5.7 with $k = m$). Selecting $m-1$ pivot elements has an external complexity of $O(m)$. Since sorting is done internally it has no external complexity. The total work of selecting pivots is :

$O(m \ \sum_{i=0}^{h} m^i), h = \log_m n - 1$

$= O(m \frac{m^{\log_m n - 1 + 1} - 1}{m-1})$, (Appendix A Proposition A.3)

$= O(n)$.

O(n) blocks are read and written at each recursion level, giving an external complexity of $O(n \log_m n)$. The total external complexity is $O(n + n \log_m n) = O(n \log_m n)$. □

5.3 Heapsort

The basic scheme in heapsort is to insert the records to be sorted into a priority queue, and subsequently extract records so that records are selected in a sorted order. The sorting complexities therefore depend on the complexities of operating on the priority queue. In chapter 4 we presented priority queues which can be used in a heap sorting algorithm. Both the internal and the external complexities are in general established as: $complexity_{sort} = N(complexity_{insert} + complexity_{delete-max})$.

5.3.1 d-heap heapsort

We choose $d = B$ since it gives us the optimum external complexity. Using the above equation, we get the internal complexity of $O(BN \log_B N)$ and the external complexity $O(N \log_B n)$.

5.3.2 B-tree heapsort

Inserting N records in a B-tree has an internal complexity of $\theta(N \log_2 N)$ and an external complexity of $\theta(N \log_B n)$ It is not necessary to execute the delete-max operations. Instead one can traverse the tree in-order, giving the internal complexity of $\theta(N)$, and the external complexity of $\theta(n)$. Therefore one can sort with an internal complexity of $\theta(N \log_2 N)$ and with an external complexity of $\theta(N \log_B n)$.

5.3.3 I/O-queue heapsort

Using the above equation, we get the internal complexity of $O(N \log_2 N)$ and the external complexity $O(n \log_m n)$.

5.3.4 Buffer tree heapsort

Like with the B-tree heapsort one can extract all N records of the buffer tree with the I/O-cost $\theta(n)$. All the buffers are emptied from top to bottom so that all records are pushed towards the leaves. The records are then extracted by scanning the leaves. Inserting the N records has an external complexity of $O(n \log_m n)$, and an internal complexity of $O(N \log_2 N)$. Inserting is therefore the expensive part of sorting with buffer trees.

5.3.5 Hillsort

In [43] Wegner and Teuhola presented an external heapsort which does not follow the principle of inserting and removing records one at a time. The algorithm rather builds a tree-structure called a hill, and then removes whole blocks of size B. A hill consists of blocks in a tree structure, each block contains B records in a sorted order. Each internal node has two children. The hill-condition (i.e. the heap-condition) states that the records of the children are all less than or equal to all parent records. The internal complexity of sorting N becomes $O(N \log_2 N)$, while the external complexity is $O(nlog_2n)$.

5.4 Related work

The general lower bound on the I/O-complexity of permuting ([4]), may in some special cases be broken. For example in matrix transpose. Other classes are identified and analysed in [15].

Many of the models presented in Section 2.5, give lower bounds of sorting, different from the lower bound in our model. The articles [2] (HMM_f), [3] (BT_f) and [45] ($P-HMM$), all give lower bounds for sorting.

Mergesort on parallel disks has an inherited problem in distributing the merged runs on the disks, in such a way that all the disks are used efficiently. [35] gives three paradigms for solving this problem in general. [8] solves the problem by randomization, and shows that the algorithm is excellent both in theory and practice.

Quicksort is tested in practice in [28] and [41]. The quicksort presented in [41], is optimized with respect to the space-time integral, which is the product of the working set size and the execution time. In [28] it is justified that quicksort is better than Distributive Partitioned Sort in practice, and it is shown that the working set for quicksort is also better.

An external version of bubblesort is presented in [19]. Experiments show that in some cases this sorting method is better than 2-way mergesort.

5.5 Conclusion

This section summerizes this chapter by comparing the different sorting algorithms we have studied, see table 5.1. We first consider the internal sorting algorithms. The external

	Internal complexity	External complexity
Lower bounds	$\Omega(N \log_2 N)$	$\Omega(n \log_m n)$
Internal mergesort	$\theta(N \log_2 N)$	$\theta(n \log_2 \frac{n}{m})$
External mergesort	$\theta(N \log_2 N)$	$\theta(n \log_m n)$
Internal samplesort	$O(N \log_2 N)$	$O(n \log_2 \frac{n}{m})$
External samplesort	$O(N \log_2 N)$	$O(n \log_m n)$
d-heap heapsort	$O(BN \log_B N)$	$O(N \log_B n)$
B-tree heapsort	$\theta(N \log_2 N)$	$\theta(N \log_B n)$
I/O-queue heapsort	$O(N \log_2 N)$	$O(n \log_m n)$
Buffer tree heapsort	$O(N \log_2 N)$	$O(n \log_m n)$
Hillsort	$O(N \log_2 N)$	$O(n \log_2 n)$

Table 5.1: Complexities of various sorting algorithms.

complexity for d-heap heapsort can be stated as $O(Bn\frac{\log_2 n}{\log_2 B})$, which is at least $\frac{B}{log_2 B}$ times worse than the internal samplesort and mergesort. Internal samplesort uses $\log_2 N$ extra space (for recursion stack or indices), whereas mergesort uses a buffer of size $N/2$. This could mean that internal samplesort is faster in practice.

Neither B-tree heapsort or Hillsort reaches the I/O lower bound for sorting. We have four I/O-optimal sorting algorithms which are also optimal with respect to internal complexity. The complexities do not give an answer to which implementation is the best choice. Samplesort does not split the input sequence in whole B-blocks, causing extra I/O's compared to external mergesort. To compare external mergesort and I/O-queue heapsort we count the number of m-way merges initiated. Mergesort begins with $\frac{n}{m}$ sequences which causes $\frac{n}{m^2}$ merges. In the next round $\frac{n}{m^3}$ and so on. This sum is limited by $\frac{n}{m-1}$. The m-way mergings in I/O-queue heapsort is done during refills. The leaves are never refilled. On next level the nodes are refilled m times and there are $\frac{n}{m^2}$ nodes, this gives $\frac{n}{m}$ m-way merges. The next level again has $\frac{n}{m^3}$ and they are refilled m^2 times, i.e. the number of refill is $\frac{n}{m}$. Since the height of the structure is $\log_m n$ the total number of m-way merges is $(\log_m n - 1)\frac{n}{m}$. The above arguments lead us to believe that external mergesort is better than the I/O-queue heapsort. By the same arguments one can assume that external mergesort is better than the buffer tree heapsort.

Chapter 6

Graph problems

Let G=($\mathcal{V}$,$\mathcal{E}$) be a graph. Further let V and E denote the number of vertices and edges in G, respectively. When B is the block size we use $v = \lceil \frac{V}{B} \rceil$ and $e = \lceil \frac{E}{B} \rceil$.

6.1 General PRAM simulation

In [12] Chiang analyses simulation of PRAM (Parallel Random Access Machines) algorithms, and why this is useful with respect to external algorithms. For a general introduction to PRAM algorithms see, for example, [16].

In one step of a PRAM algorithm, a processor reads data, executes a constant number of data operations locally and finally writes data back. The read and write locations are not necessarily the same. Since the actions executed by the different processors are independent, the chronological order is not important (except for the order of reads and writes). Therefore the operations can be sorted according to the memory addresses before simulating them on a single processor. In this way one might reduce the external complexity by a factor $\frac{1}{B}$ compared to the obvious complexity at the expense of sorting all operations.

The complexity of simulating one PRAM step depends on both the number of operations (number of processors) and on the size of the memory universe. Assume that we want to simulate a PRAM step which uses N processors and N^2 memory space. The complexity to sort the operations will be $O(n \log_m n)$ and the number of (potentially) different memory blocks are $O(\frac{N^2}{B})$. Therefore the I/O-complexity is $O(n(\log_m n + N))$. Simulating a whole algorithm of T PRAM steps is done with external complexity $O(Tn \log_m n)$, if the algorithm uses O(N) processors and memory space resp.

But for specific class of PRAM algorithms we can hope for better results. Many algorithms behave in such a manner that not all processors are active all the time. For example adding N numbers requires $N/2$ processors to achieve a PRAM time of $O(\log_2 N)$, but for every PRAM step the number of active processors is halved. This means that the number of processors to be simulated is also halved after every PRAM step. The same applies to the space requirements. For these type of algorithms we can state the following [12]:

Theorem 6.1 *If a PRAM algorithm after each stage, where each stage uses T steps, reduces the processors (initially N) and the number of memory cells ever used again (initially O(N)) by a constant factor, and the number of stages is $O(\log_2 N)$ then the algorithm can be simulated with an I/O-complexity of $O(Tn \log_m n)$.*

Proof The first stages are made in T steps, each step of that stage is simulated with an I/O-complexity of $O(n \log_m n)$ (when sorting). We can set up the following recurrence

$\mathcal{I}(N) = O(Tn \log_m n) + \mathcal{I}(\alpha N)$ for some $\alpha < 1$
which characterizes the number of of I/O's needed to simulate the algorithm, which is $O(Tn \log_m n)$.□
For example, if we wish to simulate the PRAM algorithm for adding N numbers, we would need $O(n \log_m n)$ I/O's, since T=1 (each stage consists of one step).

6.2 Graph searching

The problem of graph searching seems not to be easier than finding a minimum spanning forest (which is considered in the next section). We define a search to visit each edge $O(1)$ times and each vertex v_i $O(b_i)$ times, where b_i is the number of neighbours of vertex v_i.

6.2.1 Internal search

When using the simple methods of internal searching (for example breadth-first searching) we have a worst case I/O-complexity of $O(E)$. When during the search we have to determine, whether or not a given edge or vertex has been visited, we have to read in some auxiliary structure, giving $O(1)$ block reads per edge or $O(b_i)$ block reads for each vertex v_i. The total block read complexity is then $O(E)$. Writing to the auxiliary structure can also be shown to have the block write complexity $O(E)$. The external complexity is therefore $O(E)$.

6.2.2 PRAM simulation

In [13] a number of graph problems are solved using a generic PRAM algorithm, one of these problems is an algorithm to find a spanning forest, which is also a search tree for a graph. We use PRAM simulation on this PRAM algorithm, to construct an I/O-efficient algorithm for graph searching using one processor. Each of the $\log_2 V$ stages of the algorithm uses $\log_2 V$ steps, $\frac{V^2}{\log_2{}^2 V}$ processors and V^2 space. Using a generalized version of Theorem 6.1 we can find that the number of I/O's needed to simulate: $O(\log_2 V \frac{V^2}{\log_2{}^2 VB} \log_m \frac{V^2}{\log_2{}^2 VB} + \frac{V^2}{B}) = O(\frac{V^2}{B} \log_m \frac{V^2}{B})$.

But the same algorithm can be simulated more efficiently, also shown in [12], since we do not have to follow limitations implied by real parallelism. This means that we are not forced to have exclusive read and write and that finding a minimum numbered edge for each vertex does not require V processors (V simulation steps) but can be done in a number of steps less than or equal to the number of neighbours that each vertex actually has. In the above simulation the graph is represented in a $V * V$ matrix, this can be improved by representing the edges as an array of size E. After $\lceil \log_2 \frac{V}{M} \rceil$ iterations the number of vertices is M which means that the remaining vertices can fit into internal memory. Graph searching on vertices in internal memory is done by scanning the edge array and connecting all vertices in internal memory to the search tree. The total complexity is $O(\log_2 \frac{V}{M} e \log_m e + e) = O(\log_2 \frac{V}{M} e \log_m e)$.

6.2.3 Adaptive traversing

In [11] an external Depth-First-Search is presented which can be used in any systematic searching strategy. The order which the nodes are visited in is determined by the searching strategy. But the technique will work with any search order. The idea is to fill up internal memory, as the searching goes on, with information about which nodes have been visited so far, and only load from external memory blocks of neighbour lists.

Once the internal memory gets filled up the search is interrupted, and the external memory is scanned to remove all edges that point to nodes already visited. When this is done, the search can continue with the internal memory empty. The internal memory is filled up at most $O(\frac{V}{M})$ times, and each interruption costs $O(e)$. Besides that, each node causes $O(1)$ I/O's (when the first block of a neighbour list is loaded to internal memory), and finally all edges are read in an optimal manner ($O(e)$ I/O's). To sum up we have $O(\frac{V}{M}e + V + e) = O((1 + \frac{V}{M})e + V)$.

6.2.4 Conclusion

Table 6.1 summerizes the results from the graph searching section. To compare internal

	Internal complexity	external complexity
Internal search	$O(E)$	$O(E)$
PRAM simulation	not calculated	$\min(O(\frac{V^2}{B}\log_m \frac{V^2}{B}), O(\log_2 \frac{V}{M} e \log_m e))$
Adaptive traversing	not calculated	$O((1 + \frac{V}{M})e + V)$

Table 6.1: Complexities of various graph searching algorithms.

search with PRAM simulation we examine in which situations the external complexity of internal search is better than PRAM simulation, that is:

$$E < \log_2 \frac{V}{M} e \log_m e$$

$$\Rightarrow B < \log_2 V \log_m V$$

$$\Rightarrow V > 2^{\frac{B \log_2 m}{2}}$$

Since this is unrealistic, PRAM simulation is better in any practical application. Comparing PRAM simulation and adaptive traversing shows that PRAM simulation is better than the two. This can be shown when assuming that $\log_m e$ is a very small number, leading to an external complexity of the PRAM simulation of $e \log_2 \frac{V}{M}$. The result comes from observing that the external complexity of adaptive traversing has a $e\frac{V}{M}$ factor. Since PRAM simulation makes an arbitrary search, one has to choose one of the other methods if a more specific search strategy is needed.

The external complexity of internal search, $O(E)$, dominates the external complexity of adaptive searching, $O(\frac{V}{M}e + e + V)$, as long as $\frac{V}{M}e = O(E) \Rightarrow \frac{V}{M} = O(B)$. The choice of algorithms clearly depends on the three factors V, M and B.

6.3 Minimum spanning trees

The spanning-tree problem can be defined as follows. For an undirected, weighted graph with V vertices and E edges, select V-1 edges so that there is a path from each vertex to each other, using only the selected edges. The minimum-spanning-tree (MST) problem is that of selecting these edges so that the total weight of the selected edges is minimized. The presented algorithms assume that the graphs are connected, but can easily be extended to also process not connected graphs.

In [32] several MST-algorithms are evaluated in practice. Their main conclusion was that the Prim's and Kruskal's algorithms work best. The ranking of the two algorithms is that Prim's in most cases is the best and besides that it is the simpler of the two. These algorithms have the internal complexity $O(E \log_2 V)$ (which is asymptotical worse than the other algorithms compared in [32]).

6.3.1 Prim's algorithm

The basic idea in Prim's algorithm is to expand the set of connected vertices by one, by following the edge with the smallest weight which connects a new vertex. Initially, the set of connected vertices contains only one vertex (it does not matter which). Figure 6.1 shows how the algorithm works. Prim's algorithm can be implemented by using a priority

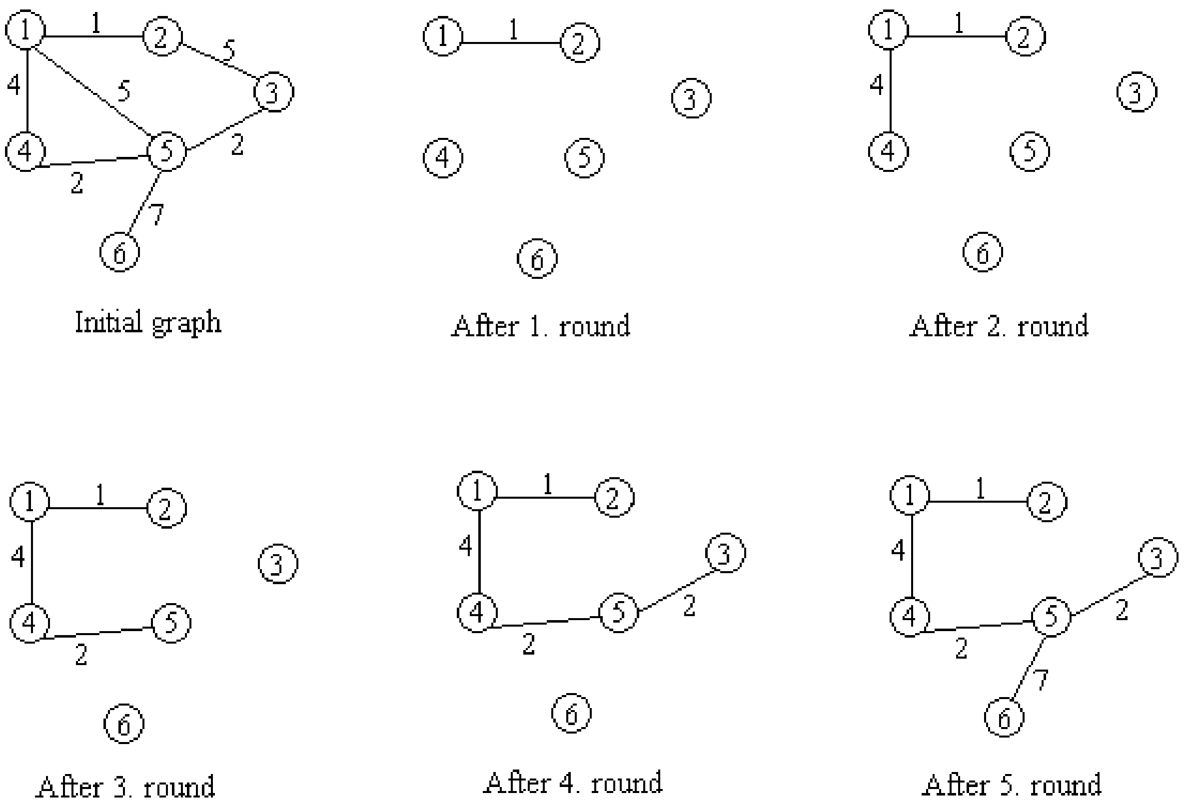

Figure 6.1: Prim's algorithm

queue containing vertices. Initially, the priority queue contains every vertex except the one we start with, that is: the priority queue contains the vertices not already connected. The initial priorities are set to infinite. Then we change the priority of all the vertices connected to the start vertex to the weight of the edges connecting it. Now we can expand the set of vertices connected, by removing the minimum record from the priority queue. Then we change the priority of the vertices connected to this node, that is, if the priority decreases. The algorithm is shown below.

```
Prim(G : Graph)
   AV = Some vertex in G
   make priority queue P
   For all vertices v except AV
     priority(v)=infinite
     v.key=priority(v)
     insert(v, P)
   repeat
     for all edges (AV,v) from AV to v
       if weight(AV,v) < priority(v) and v in P
```

```
          priority(v) = weight(AV,v)
          Decrease-key(v, priority(v), P)
      AV = Delete-min(P)
      add (AV,v) to MST
    until AV = null
    output MST
```

As we can see above, we use all three types of operations on priority queues, insert, delete-min, and decrease-key. It will be a natural choice to use the I/O-queue, presented in Section 4.3 (increase-key is substituted by decrease-key).

Theorem 6.2 *The internal complexity of Prim's algorithm is $O(E \log_2 V)$.*

Proof The algorithm starts by making V inserts. Since the algorithm stops when AV = null, we make V delete-min operations. Since we only change priority when processing an edge, we make at most E decrease-keys. When referring to the complexities in Section 4.5, we get at total complexity of $O(V \log_2 V + V \log_2 V + E \log_2 V) = O(E \log_2 V)$, since E dominates V □

Theorem 6.3 *The external complexity of Prim's algorithm is $O(e \log_m v + E)$.*

Proof As seen in the proof of Theorem 6.2, we make V inserts, V delete-min operations, and E decrease-keys. Before executing the decrease-key operations the algorithm makes at most two I/O's when comparing priority(v) to weight(AV, v) and when checking if v is in the priority queue giving an I/O-complexity of $O(E)$. When referring to the complexities in Section 4.5 (I/O-queue with increase-key), we get a total complexity of $O(v \log_m v + v \log_m v + e \log_m v + E) + O(E) = O(e \log_m e + E)$. Again we use that e dominates v. □

6.3.2 Kruskal's algorithm

In Kruskal's algorithm we process the edges in sorted order, starting with the smallest. An edge is selected for the MST, if it connects two vertices which were not connected before. Figure 6.2 illustrates the method. To implement Kruskal's algorithm, we need to use two things: sorting as presented in Chapter 5 and disjoint-sets as in [39]. The algorithm is presented below:

```
Kruskal(G : Graph)
  Sort the edges of G according to their weights
  repeat
    (v,w) = next edge in the sorted list
    if Find(v) <> Find(w)
      Union(v,w)
      Add (v,w) to MST
  until V-1 edges are found
  output MST
```

Theorem 6.4 *The internal complexity of Kruskal's algorithm is $O(E \log_2 E)$.*

Proof The sorting has an internal complexity of $O(E \log_2 E)$. At worst, we have to inspect all edges in the sorted list, this means that we make $O(E)$ find operations. The algorithm stops, when we have made $O(V)$ unions. As shown in [39], the internal complexity of making E find operations and $O(V)$ union operations is $O(E\ \ \alpha(E, V))$, where α is the inverse Ackermann's function. This gives us the wanted result of $O(E \log_2 E + E\ \ \alpha(E, V)) = O(E \log_2 E)$. □

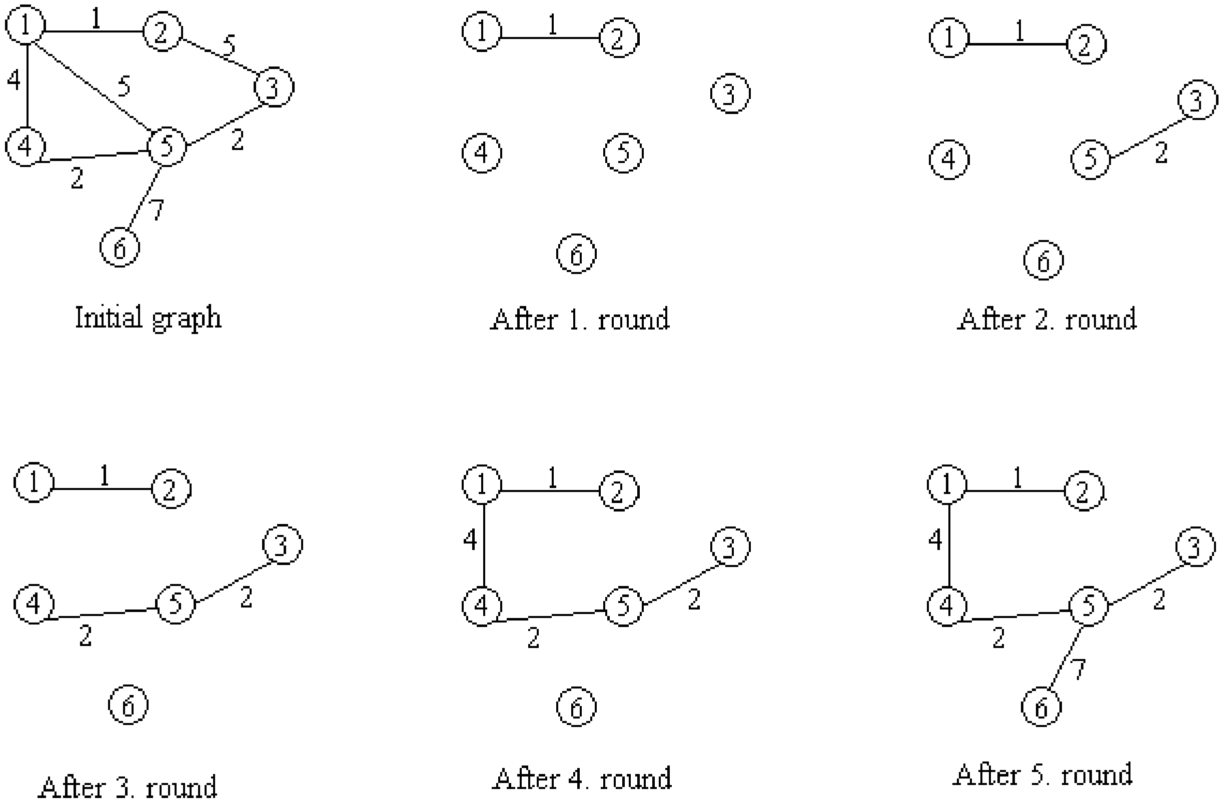

Figure 6.2: Kruskal's algorithm

Theorem 6.5 *The external complexity of Kruskal's algorithm is* $O(e \log_m e + E \;\; \alpha(E, V))$, *where* α *is the inverse Ackermann's function.*

Proof As seen in Chapter 5, the sorting can be done with $O(e \log_m e)$ I/O's. If we study the way the disjoint-set are implemented in [39], we see that in the worst-case it is necessary to make as many I/O's as memory accesses. That is, the total external complexity is $O(e \log_m e + E \;\; \alpha(E, V))$. $\square$

6.3.3 Kruskal's algorithm with a buffer

In this section we will present a variant of Kruskal's algorithm, which buffers the operations. As we shall see, we gain on the external complexity but the internal complexity gets worse. As in the standard Kruskal algorithm, we begin by sorting the edges. The basic idea is to scan $\frac{M}{6}$ edges at the time, moving them to the internal memory. In internal memory, each edge occupies 6 records, 3 for each vertex it connects. The first record is the vertex number and the second and third indicates an old and a new set number. Figure 6.3 shows the configuration of both the internal and external memory. First the next $\frac{M}{6}$ edges to be processed are loaded in internal memory. After this, all the vertices are scanned. At each vertex we scan the internal memory, filling both the old and new set number fields with the right values. Then we execute the find and union operations in internal memory, that is: if the two vertices connected by an edge, have different new set numbers, we change one of the new set numbers to the other. This change is not only made in the current vertex, but in all vertices in internal memory which have the same new set number. After executing this for every edge in internal memory, we scan the external

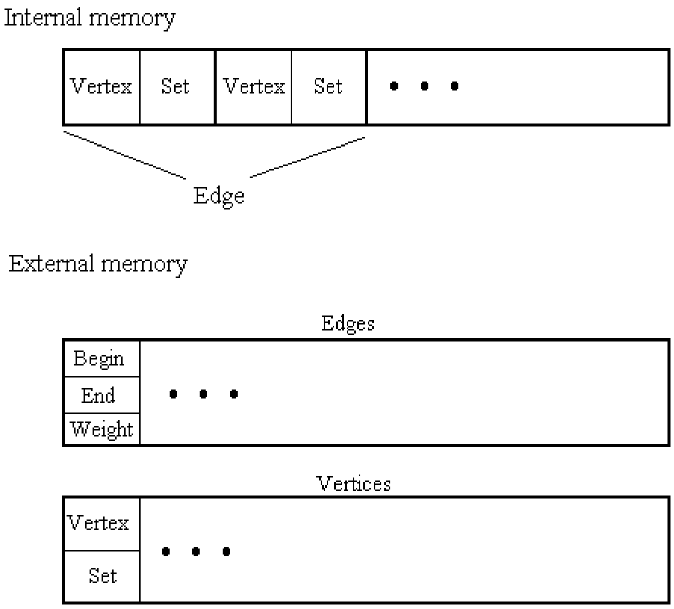

Figure 6.3: Memory configuration

vertices once again and for each vertex changing its old set number to a new set number if necessary. We continue with the next $\frac{M}{6}$ edges. The algorithm is presented below.

```
ExternKruskal(G : Graph)
   Sort the edges
   repeat
     copy next M/6 edges to internal memory
     for V1 = all vertices in external memory
       for V2 = all vertices in internal memory
         if V1 = V2
           V2.oldset = V2.newset = V1.set
     for (V1,V2) = all vertex-pairs in internal memory
       if V1.newset <> V2.newset
         Add edge (V1,V2) to MST
         s=V2.newset
         for V3 = all vertices in internal memory
           if V3.newset = s
             V3.newset = V1.newset
     for V1 = all vertices in external memory
       for V2 = all vertices in internal memory
         if V1.set = V2.oldset
           V1.set = V2.newset
   until V-1 edges are found
   output MST
```

Theorem 6.6 *The internal complexity of Kruskal's algorithm with a buffer is $O(E(V + M + \log_2 E))$.*

Proof The initial sorting has an internal complexity of $O(E \log_2 E)$. We start the repeat-until loop by filling the internal memory with vertices. This costs $O(M)$. Then we run through the vertices and scan the internal memory at each vertex. This costs $O(VM)$. It costs $O(M^2)$ to do the find and union operations. Again, it costs $O(VM)$ to change the set numbers of the vertices in external memory. The repeat-until loop is executed at most $\frac{E}{\frac{M}{6}}$ times, giving us a total internal complexity of $O(E\log_2 E + \frac{6E}{M}(M + VM + M^2 + VM)$ which leads to the claimed result. □

As one can see, we have not tried to improve the internal complexity, this could possibly be done.

Theorem 6.7 *The external complexity of Kruskal's algorithm with a buffer is* $O(\frac{ev}{m} + e\log_m e)$.

Proof The initial sorting has an external complexity of $O(e\log_m e)$. Every filing of internal memory uses $O(m)$ I/O's. We scan the vertices two times, once when inserting the set numbers into internal memory and once when writing them back. This gives $2v$ I/O's. The find and union operations give us no I/O's. Since the repeat-until loop is executed at most $\frac{E}{\frac{M}{6}}$ times, the total complexity is $O(e\log_m e + \frac{6E}{M}(m + 2v))$, which leads to the claimed result. □

6.3.4 PRAM simulation

The algorithm simulated in Section 6.2 can also be used to find minimum spanning trees with the same complexities. The two different simulations give us the upper bounds on the external complexity of $\min(O(\frac{V^2}{B}\log_m \frac{V^2}{B}), O(\log_2 \frac{V}{M} e\log_m e))$.

6.3.5 Conclusion

Table 6.2 sums up the different MST-algorithms studied above. Prim's and Kruskal's

	Internal complexity	External complexity
Prim's	$O(E\log_2 V)$	$O(e\log_m v + E)$
Kruskal	$O(E\log_2 E)$	$O(e\log_m e + E\,\alpha(E,V))$
Kruskal with buffer	$O(E(V + M + \log_2 E))$	$O(\frac{ev}{m} + e\log_m e)$
PRAM simulation	not calculated	$\min(O(\frac{V^2}{B}\log_m \frac{V^2}{B}), O(\log_2 \frac{V}{M} e\log_m e))$

Table 6.2: Complexities of various MST algorithms.

algorithms are of the same magnitude. To decide which of the two Kruskal algorithms is the better we compare the external complexities of Kruskal's algorithm to Kruskal's algorithm with a buffer:

$$e\log_m e + E\ \ \alpha(E,V) < e\log_m e + \frac{ev}{m}$$
$$\Rightarrow E < \frac{ev}{m}$$
$$\Rightarrow B < \frac{V}{M}$$
$$\Rightarrow V > BM$$

This means the external Kruskal is the best of the first three algorithms when the number of nodes in a graph are not extremely huge. Assuming that $\log_m e$ is very small the comparison of Kruskal's algorithm with a buffer and the PRAM simulation is the following:

$$\frac{ev}{m} + e\log_m e > \log_2 \frac{v}{m} e\log_m e$$
$$\Rightarrow \frac{v}{m} > log_2 \frac{v}{m}$$

This is always the case, i.e. the PRAM simulation is the most I/O-efficient MST-algorithm.

6.4 Related work

In [11] lower bounds for several graph problems are established for the first time. These lower bounds are matched with algorithms with worst case complexities. Besides a number of general techniques are developed.

The problem of searching external graphs, is described in [34]. Classes of trees and graphs are defined and analysed, with respect to storage and searching. Redundancy (blocking) is used to achieve optimal results.

The possibility of deducing information on external storage, by using the information in internal storage, is examined in [20]. Especially the graph connectivity and path finding problems are considered.

Practical experiments on maximum flow, minimum cost flow, assignment problem and shortest path, are summarized in [22].

Part II

Implementation

In this part we have practical evaluation on some of the work done in the theoretical part. We implement selected solutions to examine the practicality and to get an idea of the magnitude of the asserted improvements. We have chosen to implement sorting and priority queues because of simplicity and because we wanted to try out our I/O-queue structure. We choose only to consider the I/O-complexities, and do not evaluate the internal complexities of the different algorithms.

Chapter 7

Simulating external and virtual memory

We have chosen to analyse the performance of the algorithms using simulators, instead of trusting a operating system. This is done for several reasons: We can choose what resources to monitor, it is easier to implement algorithms not using the operating systems memory management (for example avoiding forced page replacements and shared internal memory), and we can vary parameters such as block size and internal memory size. In Appendix B the simulator programs are listed.

We have implemented two simulators. The first simulates virtual memory, the second external memory. The virtual memory simulates the "least recently used" page replacement algorithm (LRU), as described in [37]. This means that a programmer can use the memory as were it an array. The simulator keeps track of which blocks are in internal memory, and counts the number of I/O's performed. The simulator always writes back a block to external memory when it is forced out of internal memory, even if the block has not been changed since it was read. This is of some importance for some of the following algorithms. For a print of the virtual simulator see Appendix B file *VirMem.h*.

The external memory simulator leaves all the work to the programmer. The programmer can access the internal memory as a normal array, but he has to use the functions BlockRead and BlockWrite to move block between internal and external memory. This simulator also counts the number of I/O's excecuted. See Appendix B file *ExtMem.h* for a print of the external simulator.

The simulators only monitor the data accesses and not the program stack. This means that the recursion costs and cost of referencing local variables are not added to the number of I/O's.

Chapter 8

Implementing sorting

In this chapter we will present the results of executing some sorting algorithms. We have chosen sorting, because it is a simple problem and because we can test many of the algorithms described in the previous chapters. We implement an internal and an external version of quicksort, mergesort, B-tree heapsort and heapsort respectively.

To present the results we generate a number of charts, where the I/O is a function of N. Each line of the charts illustrates an execution with fixed m's and B's. We will make a chart for each of the 8 algorithms implemented with 12 combinations of m and B. We also have a chart comparing the internal and the external versions of the 4 sorting strategies, with 3 combinations of m and B for each implementation. Finally, we compare the four different external algorithms. The sorting programs are listed in Appendix C

8.1 Mergesort

The internal mergesort is programmed as an ordinary implementation, using recursion and 2-way merging. The memory management is handled by the virtual memory simulator. The external complexity could be improved, if the simulator did not write back unchanged (clean) blocks. The result is seen on figure 8.1 The I/O-complexity is $\theta(\frac{N}{B}\log_2\frac{\frac{N}{B}}{m})$. As

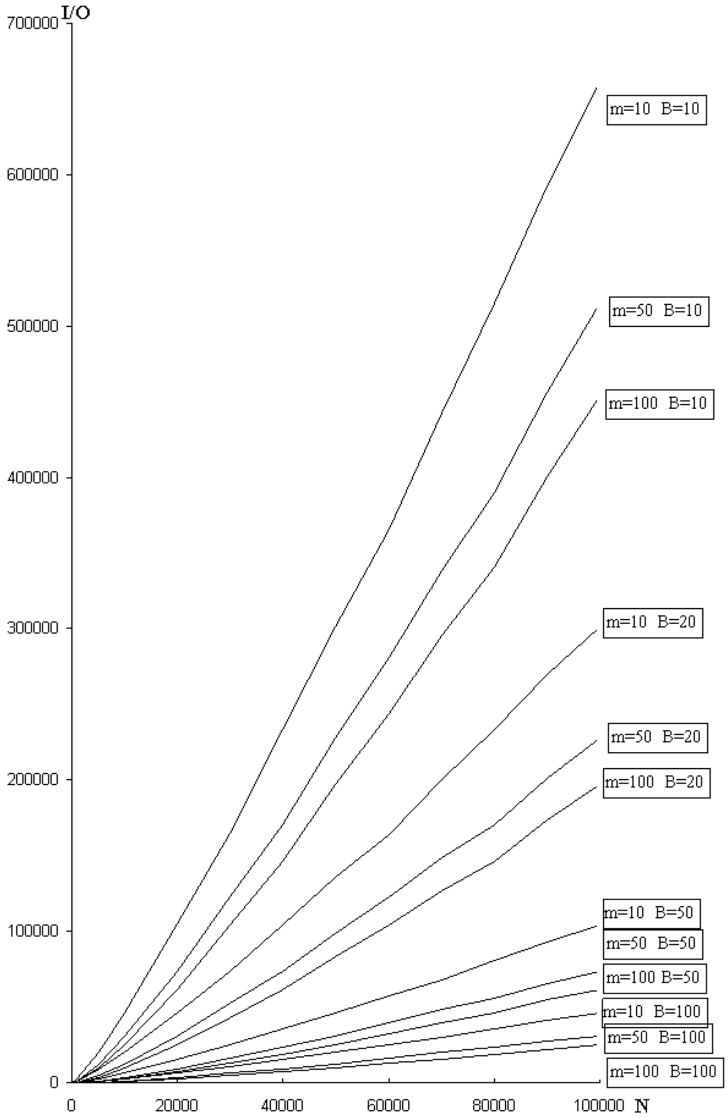

Figure 8.1: Internal mergesort in virtual memory

expected, B has more significance than m. For example, the combination $m = 100$, $B = 10$ is very much worse than the combination $m = 10$, $B = 20$, even though it has 5 times more internal memory.

The external mergesort uses an almost m-way merge, since we use 1 block as out-buffer and a number of blocks for indices for input scanning. The merge process is done bottom-up (no recursion). The process begins with internal mergesort of lists of size $\frac{2}{3}m$ blocks. The algorithm has an extra workspace equal to input size. The lists are read from an input area and merged to the workspace. In the next merge round the workspace becomes the input area etc. Results in figure 8.2. The I/O-complexity is of the form $\theta(\frac{N}{B}\frac{\log_2 \frac{N}{B}}{\log_2 m})$.

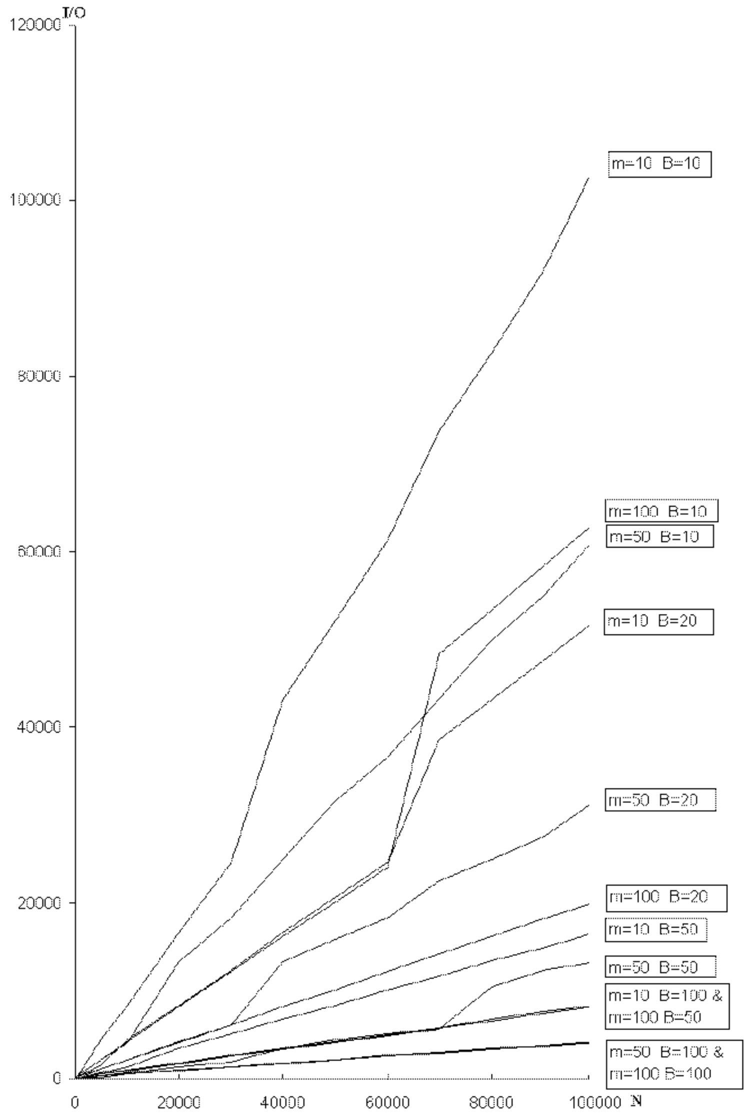

Figure 8.2: External mergesort

We can expect that m means more here than in the internal samplesort, but B is still more significant since it is nearly a linear factor and m is logarithmic. This can also be seen from the charts.

Figure 8.3 compares the two mergesort algorithms. As excepted, the external merge-

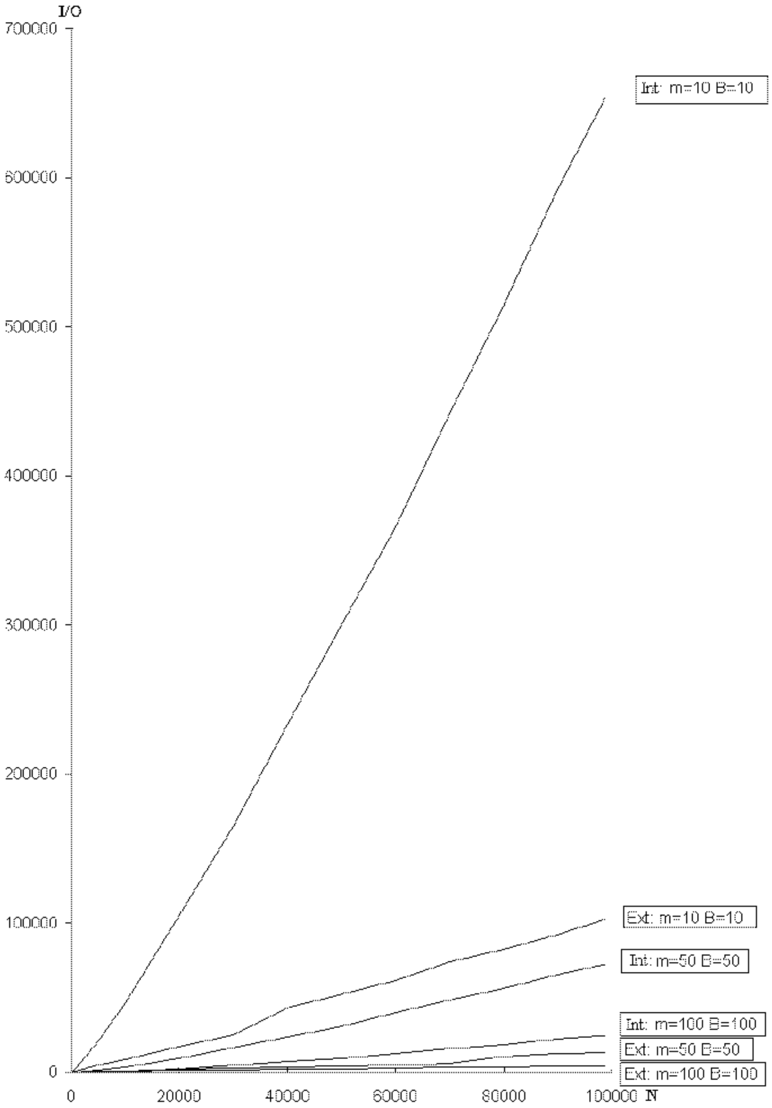

Figure 8.3: Internal and external mergesort

sort is better than the internal. The external algorithm is at bit more complicated to implement, but the gain for large N's justifies that work.

8.2 Samplesort

The internal samplesort (quicksort) makes a 2-way split on each recursion level and uses no extra space except for the recursion stack. It is executed on the virtual memory simulator since it does not administer external memory. Figure 8.4 shows the results. The I/O-

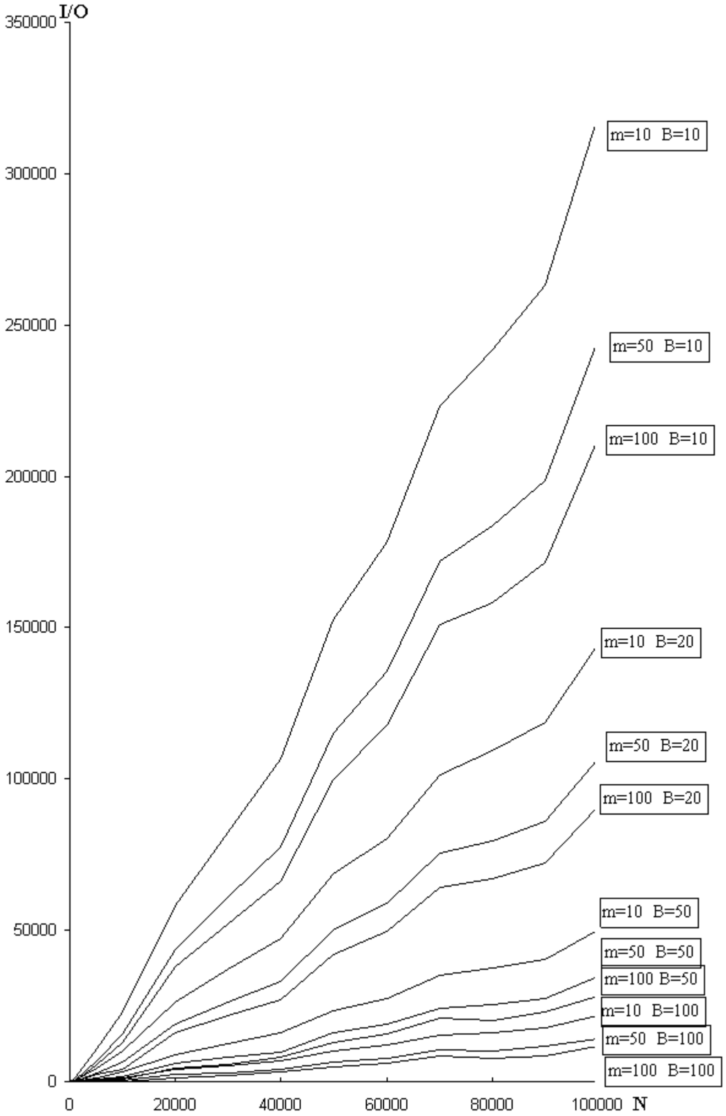

Figure 8.4: Internal samplesort in virtual memory

complexity of internal samplesort is the same as internal mergesort and one can draw the same conclusions (see Section 8.1).

External samplesort generates an almost m-way split using the first records as pivot elements. The split-lists are placed in an output area of size $\theta(N)$ and moved back to the input area before recursion. Figure 8.5 shows the results. The external complexity of

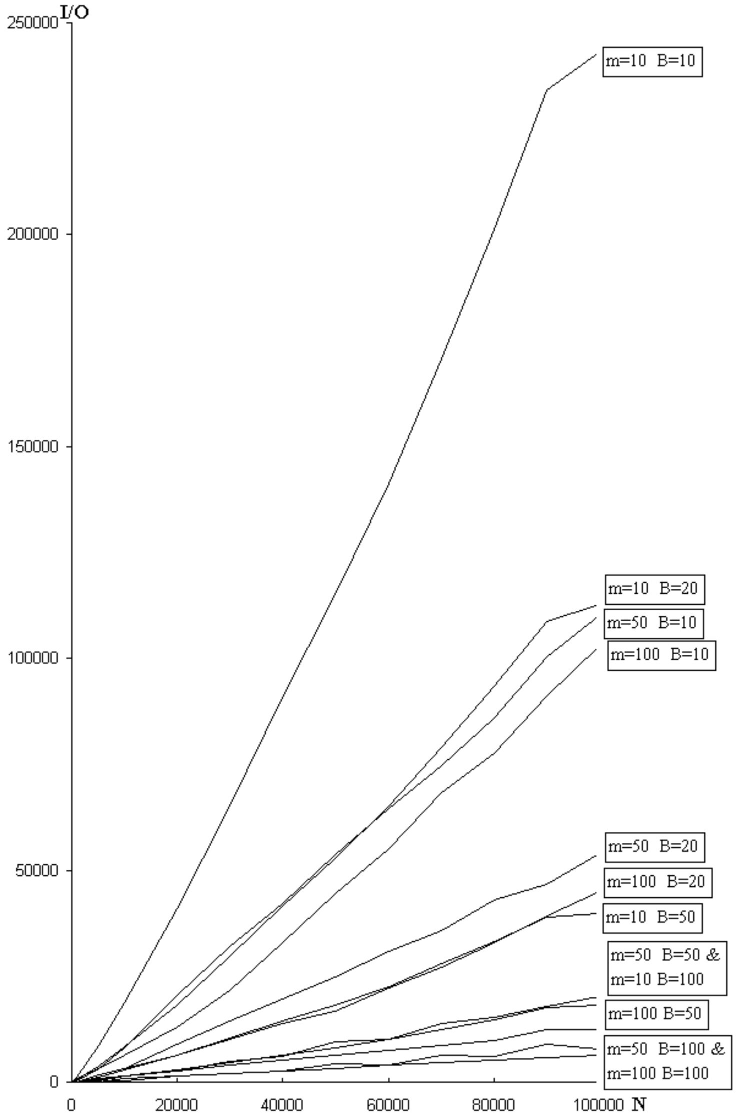

Figure 8.5: External samplesort

the external samplesort is the same as for external mergesort, and one can draw the same conclusions (see Section 8.1).

In figure 8.6 the internal and external samplesorts are compared. The external algo-

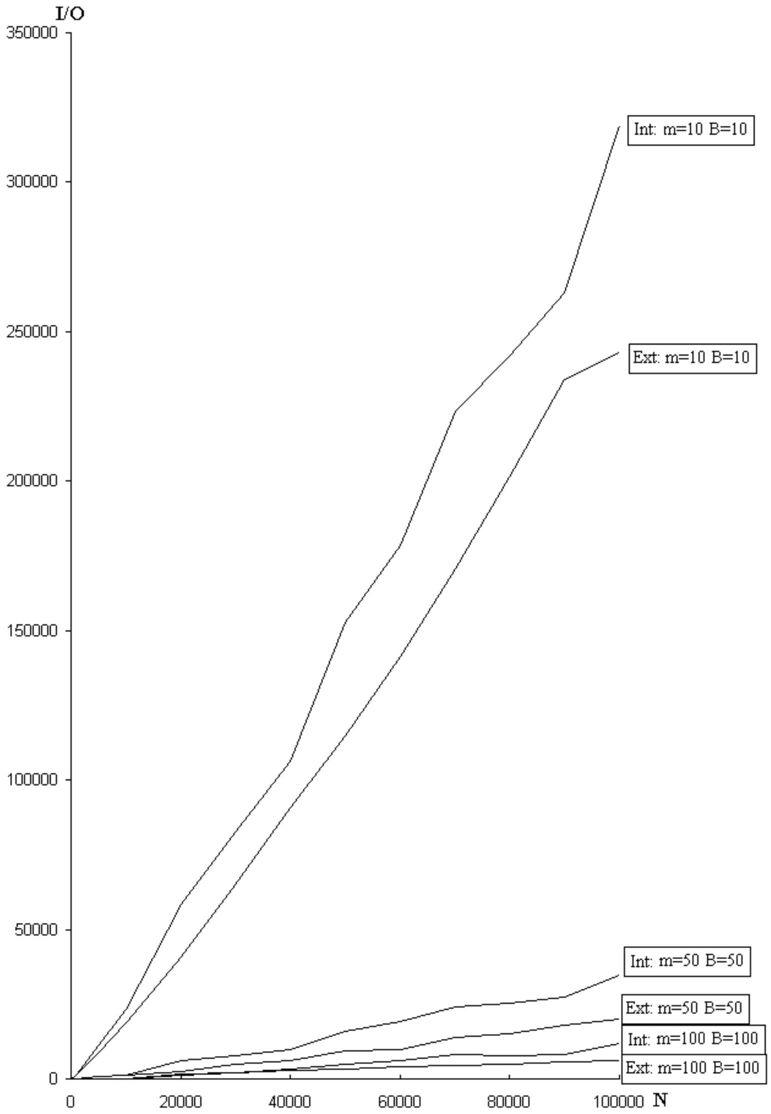

Figure 8.6: Internal and external samplesort

rithm is, in any case, better, but a lot more complicated to implement, and it uses more external memory.

8.3 Heapsort

The internal version uses a binary heap (a 2-heap). We could have chosen to implement the theoretical optimum (a B-heap), but we wish to compare the external version with a standard heapsort which we consider the binary heapsort to be. Figure 8.7 shows the results. The I/O-complexity is $O(N \log_2 \frac{N}{B})$, which leads us to believe that B is of more

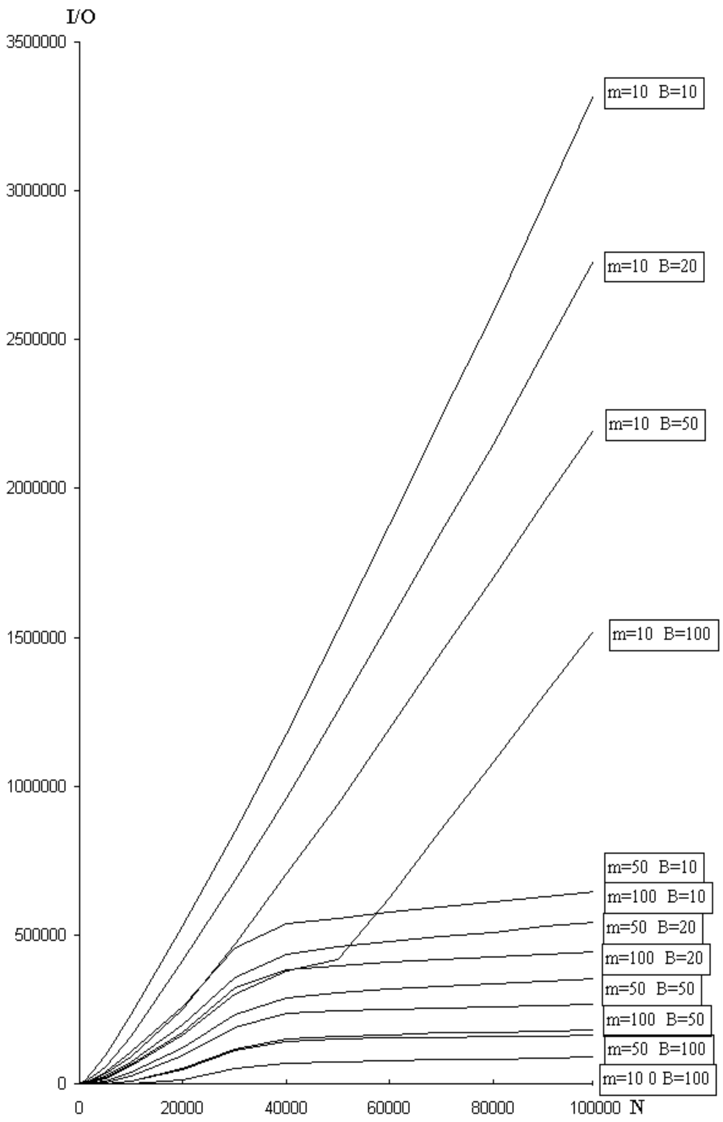

Figure 8.7: Internal heapsort in virtual memory

importance than m. But as one can see from the chart, this is not entirely the case. The lines fall in two parts. The upper part contains the combinations with small m's. This happens because the internal memory does not have place for the top of the heap as the algorithm swaps down the top record (during delete-max). When m is larger than the height (measured in blocks) $\log_2 n$, the blocks containing the top records are never replaced by the virtual memory. In this case B's importance is clear.

The external heapsort uses the I/O-queue described in Section 4.3. The fan-out is not entirely m since we only perform an almost m-way merging. When sorting it is not necessary to use the leaf melt operation since there are no insertions after delete-max operations, therefore the leaf melt operation is not implemented. The 2-way merging performed during shift-up's uses the blocks of the insert buffer (which is emptied prior to the shift-up's). The m-way merge during refill is done using the delete-max buffer, such that first the root is refilled and stored in external memory, thereafter the free space is for eventual other refill's. Figure 8.8 shows the results of the the experiments. The patterns of

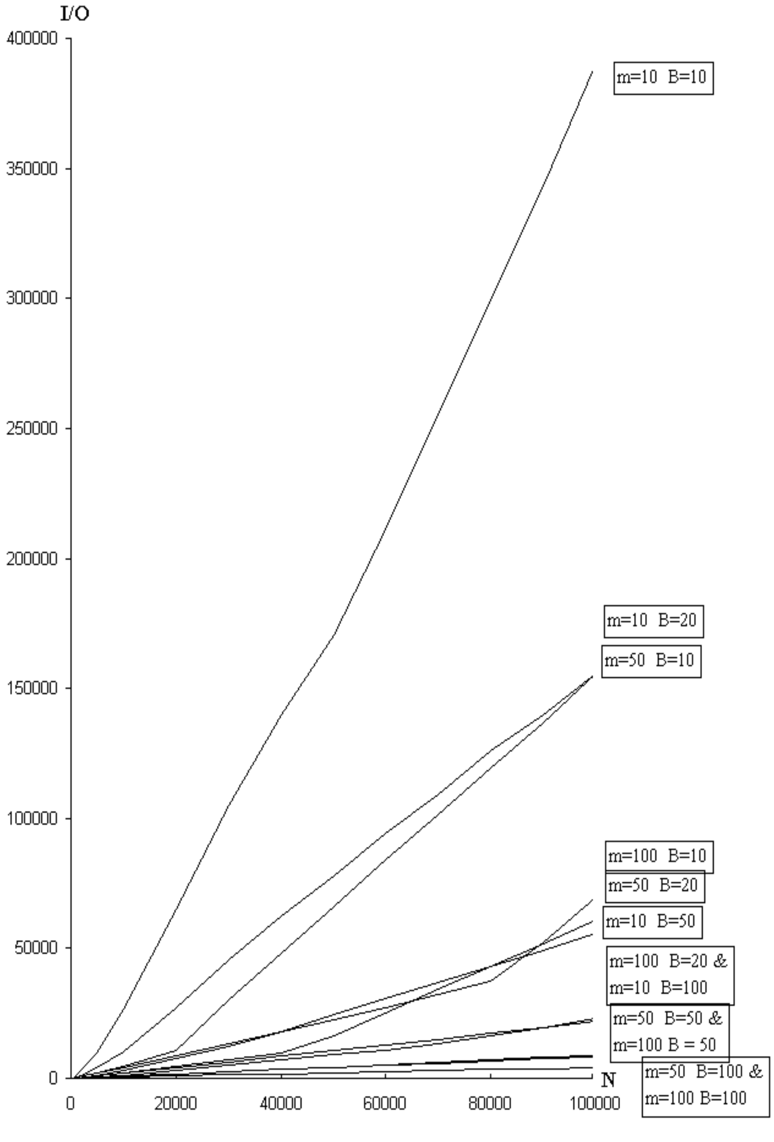

Figure 8.8: External heapsort

this optimal external algorithm is the same as with the external mergesort and samplesort, that is B dominates.

Figure 8.9 compares the two heapsorts. As expected the external one is much better.

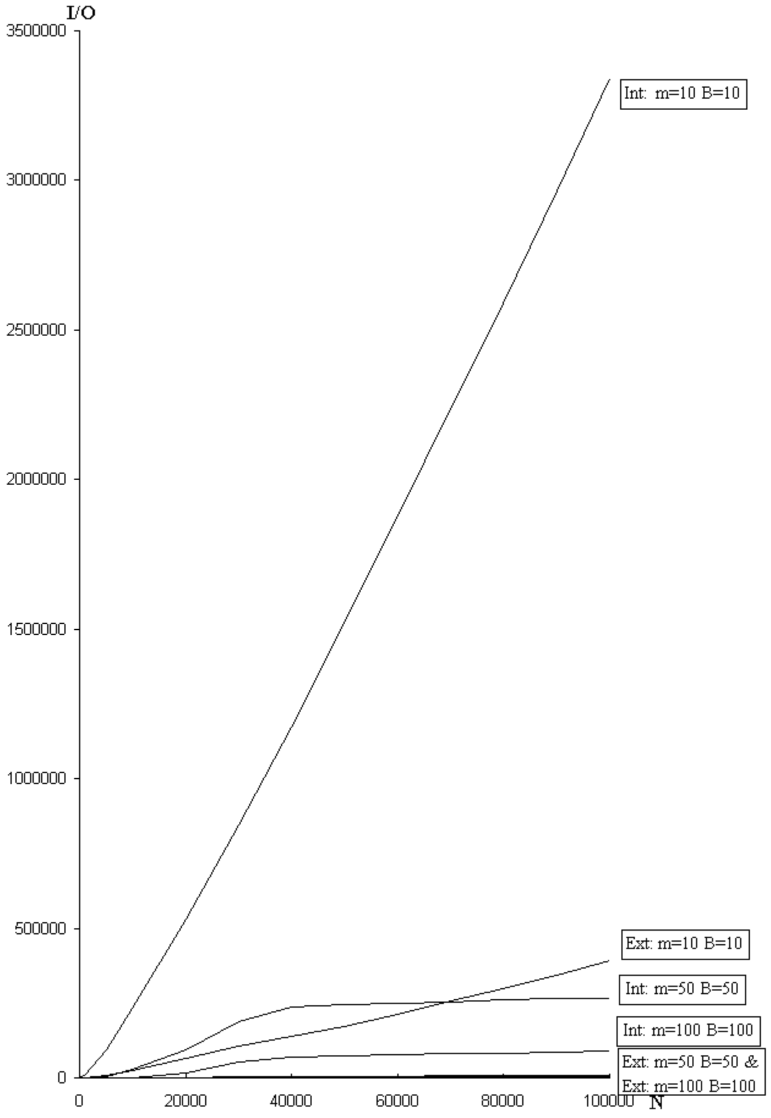

Figure 8.9: Internal and external heapsort

This huge difference between the two algorithms would be smaller if we had implemented the internal version using B-heaps. In the next chapter B-heaps are implemented when used as priority queues.

8.4 B-tree heapsort

We have chosen to implement B-tree insertions in a different way than being described in Section 3.2. The difference is that we execute the splittings on the way down, instead of traversing back (see [16]). The complexity is the same. The internal and the external versions are almost identical. The difference lies in the external version administering the external memory, whereas the internal one uses virtual memory. Figure 8.10 shows the results of the runs with internal B-tree heapsort. The I/O-complexity of the internal

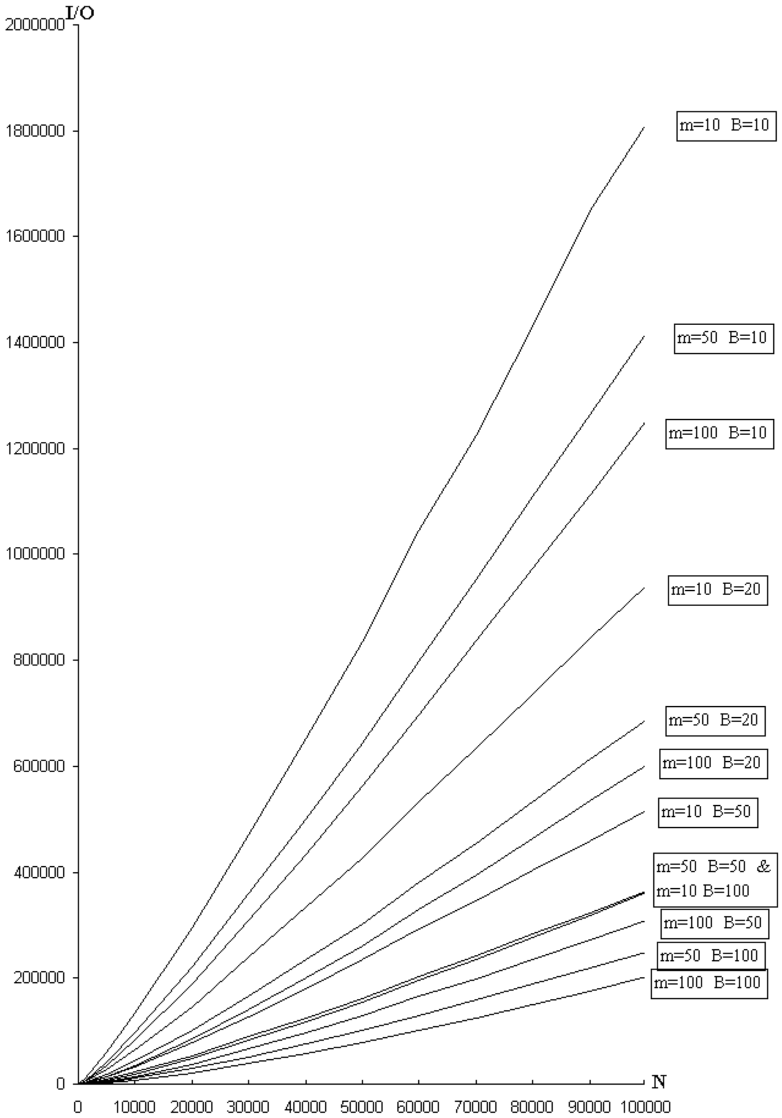

Figure 8.10: Internal B-tree heapsort

B-tree heapsort is $\theta(N \log_B \frac{N}{B})$, which fits the experimental results, where B has a big importance.

There are two differences between the internal and the external versions of B-tree heapsort. The external algorithm keeps the root resident in internal memory during the insertions. During the in-order traversing of the tree (extraction), the blocks kept are those which are to be accessed next. This means that the m lowest blocks of the path from the root to the 'current' leaf are kept internally. This is an optimal replacement strategy, since we replace those blocks which are to be used again last. Figure 8.11 shows the results. The I/O-complexity of the external B-tree heapsort is the same as in the

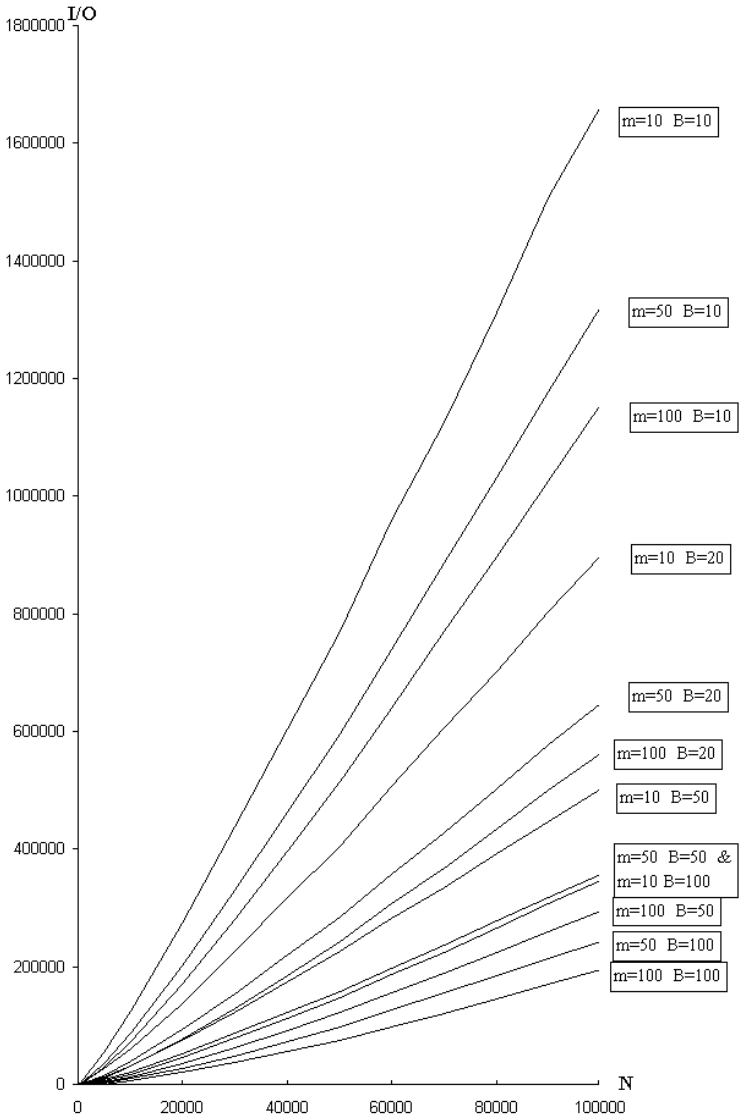

Figure 8.11: External B-tree heapsort

internal implementation and the results are similar.

Figure 8.12 shows the two algorithms compared. The improvement of the external

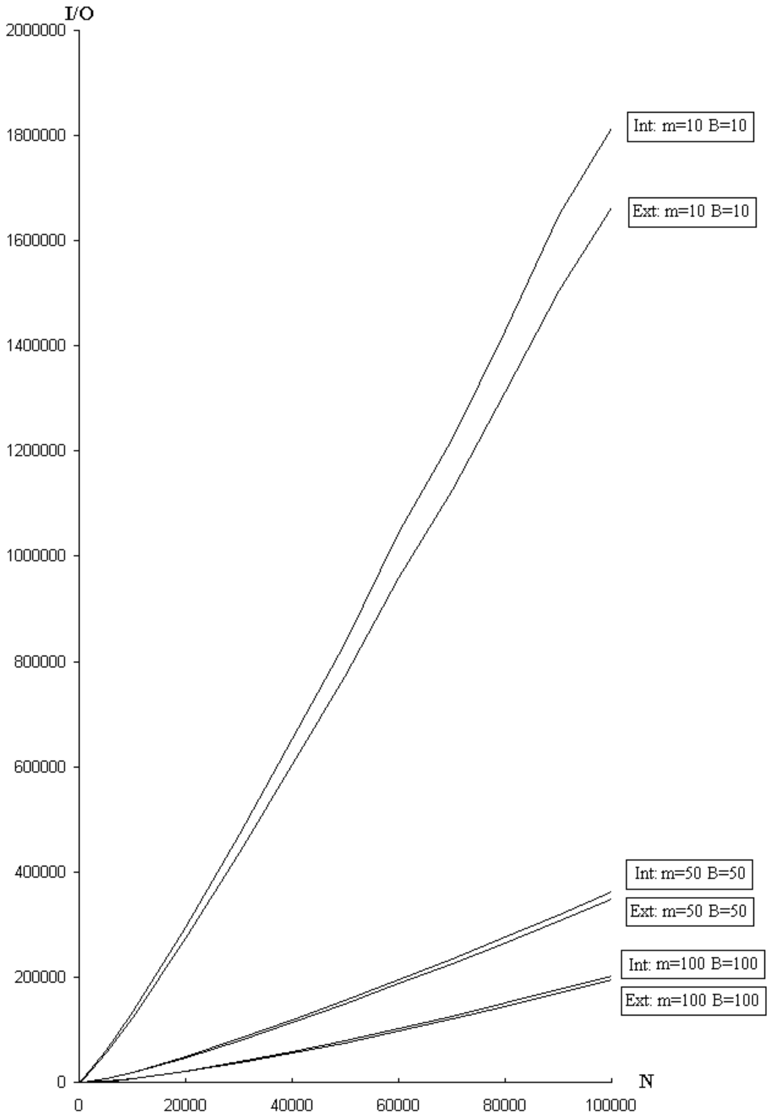

Figure 8.12: Internal and external B-tree heapsort

B-tree heapsort compared to internal B-tree sort is small and does not justify the implementation of memory management.

8.5 Comparison of the external algorithms

In the following we compare the external sorting algorithms. First, we have the four algorithms with three different combinations of m and B, for the same N's as above. After that we compare external heapsort to the commonly used external mergesort, by using larger N's. Figures 8.13, 8.14 and 8.15 compare the four external sorting algorithms.

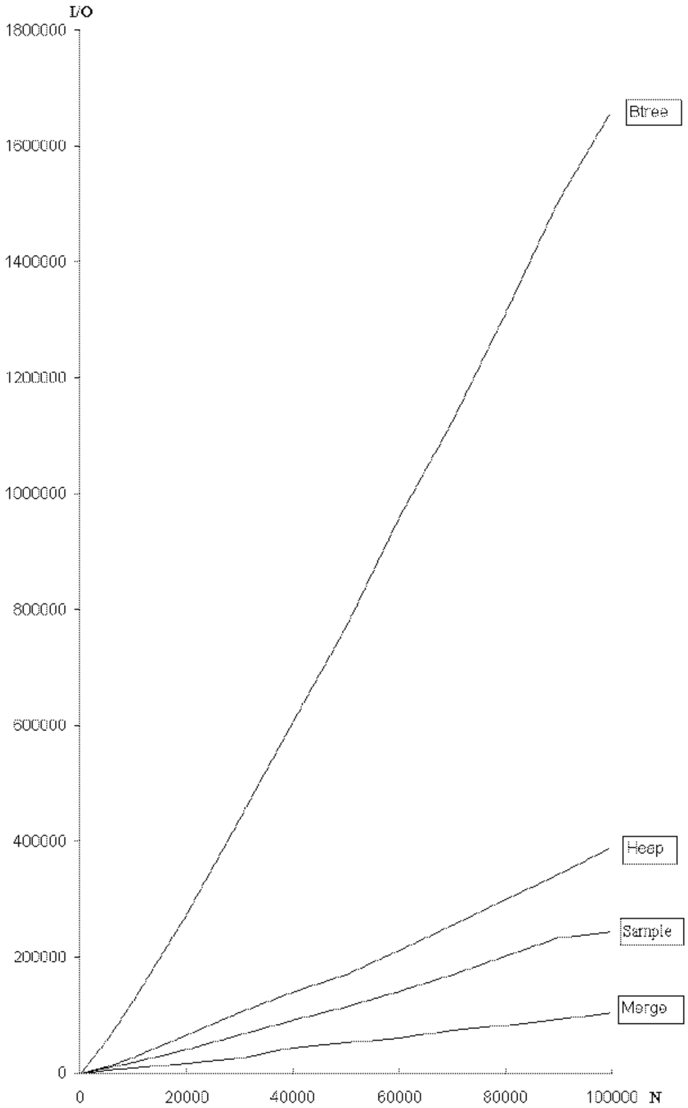

Figure 8.13: External algorithms with m=10 B=10

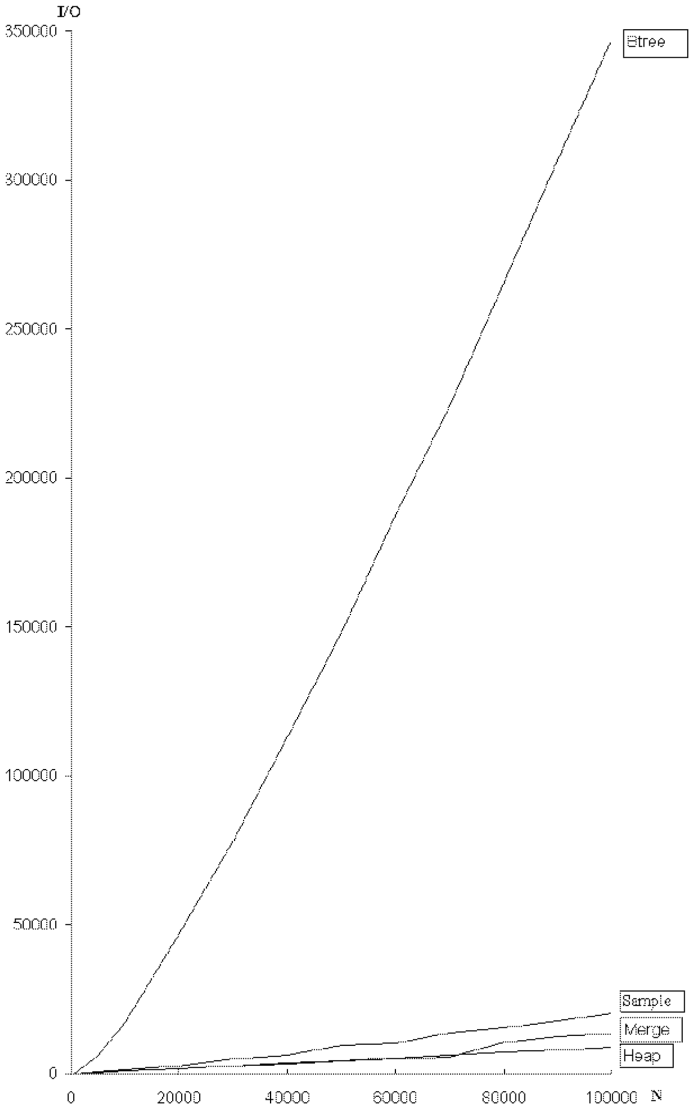

Figure 8.14: External algorithms with m=50 B=50

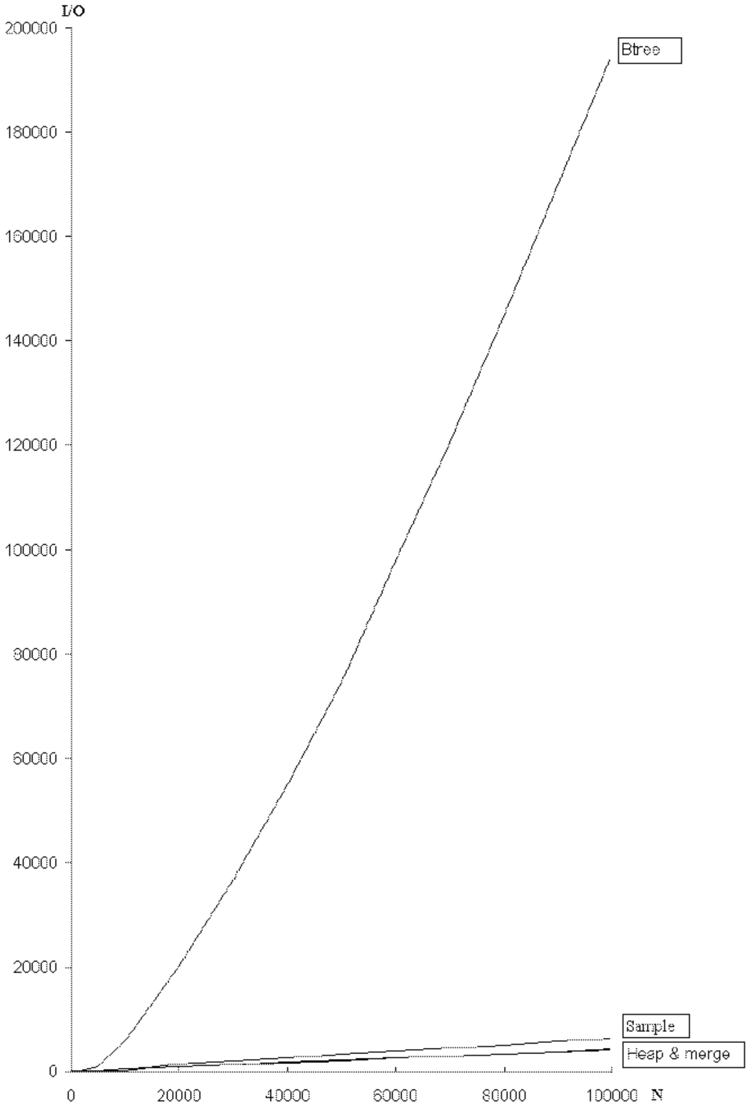

Figure 8.15: External algorithms with m=100 B=100

It is clear from the charts that B-tree heapsort is not a good idea: B-trees are designed for dictionaries, not for sorting. This also shows that our I/O-queue (the heapsort) is much better than the B-tree queue. The external samplesort seems not to be an obvious choice but it is not extremely bad. In the chart with the smallest internal memory, external merge is clearly better than heapsort, but in the other charts their performance is about the same. To examine if our heapsort is better than mergesort even for large N's, we have tested this for N up to $1,000,000$. The result is shown in figure 8.16. External mergesort

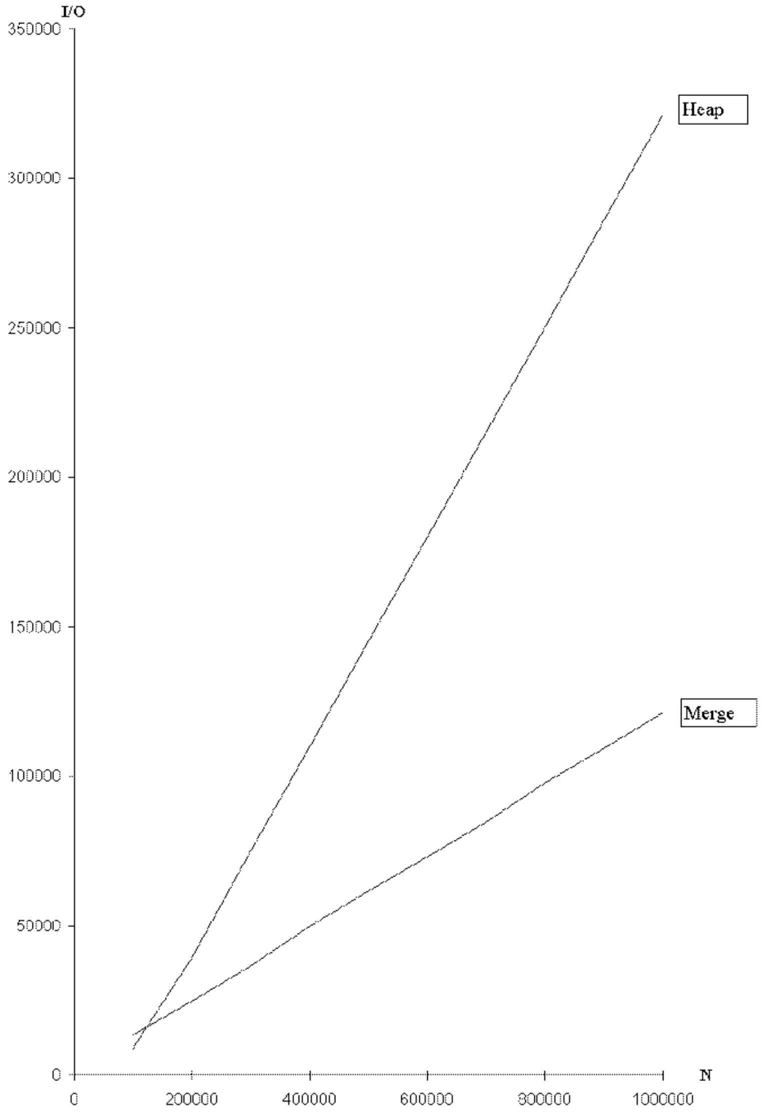

Figure 8.16: External mergesort and heapsort m=50 B=50

has about a third of the number of I/O's compared to external heapsort. This is not so surprising since mergesort is designed specially for sorting.

8.6 Conclusion

This chapter shows that the theoretical considerations of Chapter 5 *Sorting* are reflected in practice. All the external algorithms sort faster than there respective internal version, cf. the better complexities of the external algorithms. Among the external algorithms we predicted that external mergesort would be the best, which our simulation also shows.

Chapter 9

Implementing priority queues

When using the queues for sorting not all the aspects of the structures are shown. For example the insertions and the delete-max operations are not intermixed and the B-tree delete-max is not even used. Therefore we will test them as real priority queues. We have implemented three priority queues, the B-heap (a d-heap with $d = B$), the B-tree and the I/O-queue, all described in chapter 4. The only operations against the structures are insertions and delete-max operations.

The experiments begin with inserting a number of records (N insertions). Thereafter we begin counting I/O's. Now we execute N randomly chosen operations, with a ratio α between the number of insertions and the number of delete-max operations. That is if $\alpha = 25\%$, 25% of the operations are insertions. This is done for different combinations of N, α, m and B. To show the results we first have a chart for each queue, containing three different α's. Then a chart for each α comparing the three different queue's. Finally we have a chart with three different m-B combinations for each queue. Each chart shows the I/O per operation as function of the average queue size. The average queue size as calculated from the initial size N, and the α value. That is with $N = 10000$ and $\alpha = 25\%$ the expected average size is 7500. The Priority queue programs are listed in Appendix D

9.1 B-heap

The B-heap is a B-ary heap as decribed in Section 4.1. It is implemented on the virtual memory simulator. Figure 9.1 shows the results with varying α. With $d = B$ the com-

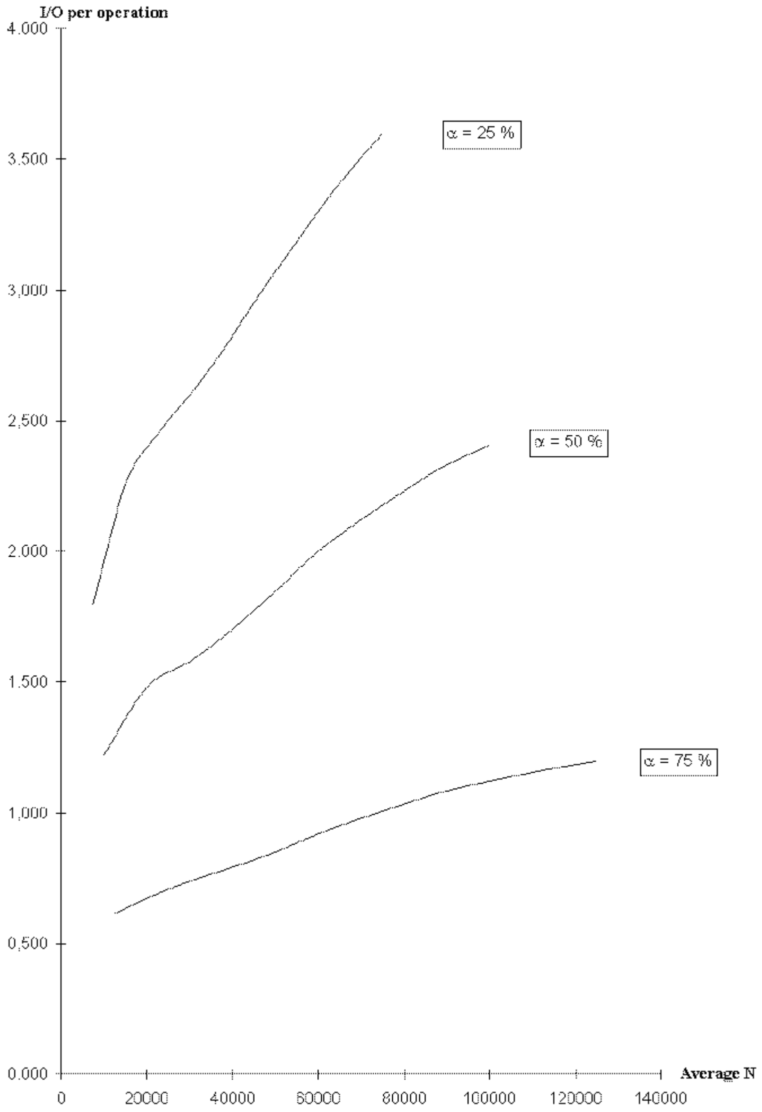

Figure 9.1: B-heap with m=50 and B=50

plexeties for the two operations are both $O(\log_B n)$, but the insertions can be expected to be cheaper since the shift-ups are not expected to swap all the way. On the other hand, the delete-max we expect to swap down nearly all the way, since a leaf record is placed in the root. This is seen from the chart to be true.

Figure 9.2 shows the B-heap with varying internal memory size. The chart shows that

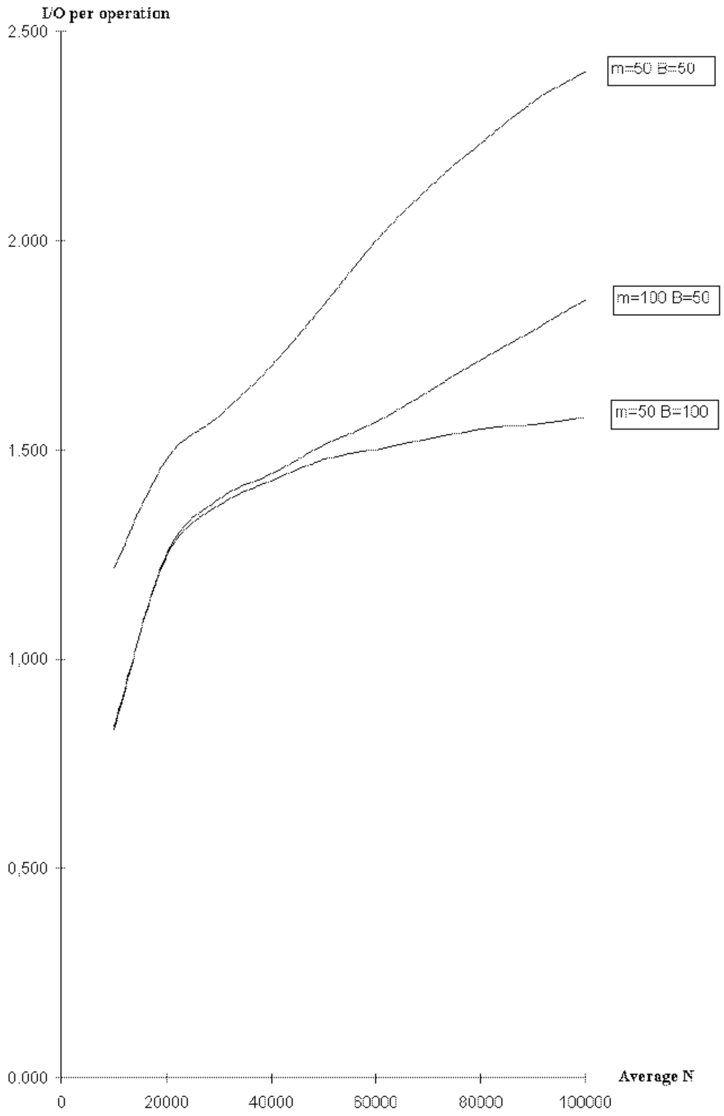

Figure 9.2: B-heap with $\alpha = 50\%$

larger internal memory gives lower I/O's (the internal memory is doubled in the two cases). It also shows that B means more than m which is not surprising.

9.2 B-tree

As described in Section 8.4, the B-tree heap is implemented so that an insertion only uses one traversal down the tree. There is implemented a delete-max buffer as decribed in Section 4.2, that is the maximum records are kept internally and the buffer is refilled when needed. Since the B-tree heapsort showed only litle improvement when using external algorithms for B-trees, we have chosen to implement it virtually. Figure 9.3 shows the B-tree priority queue whit varying α. Since the complexity for inserting is $\theta(\log_B n)$ and

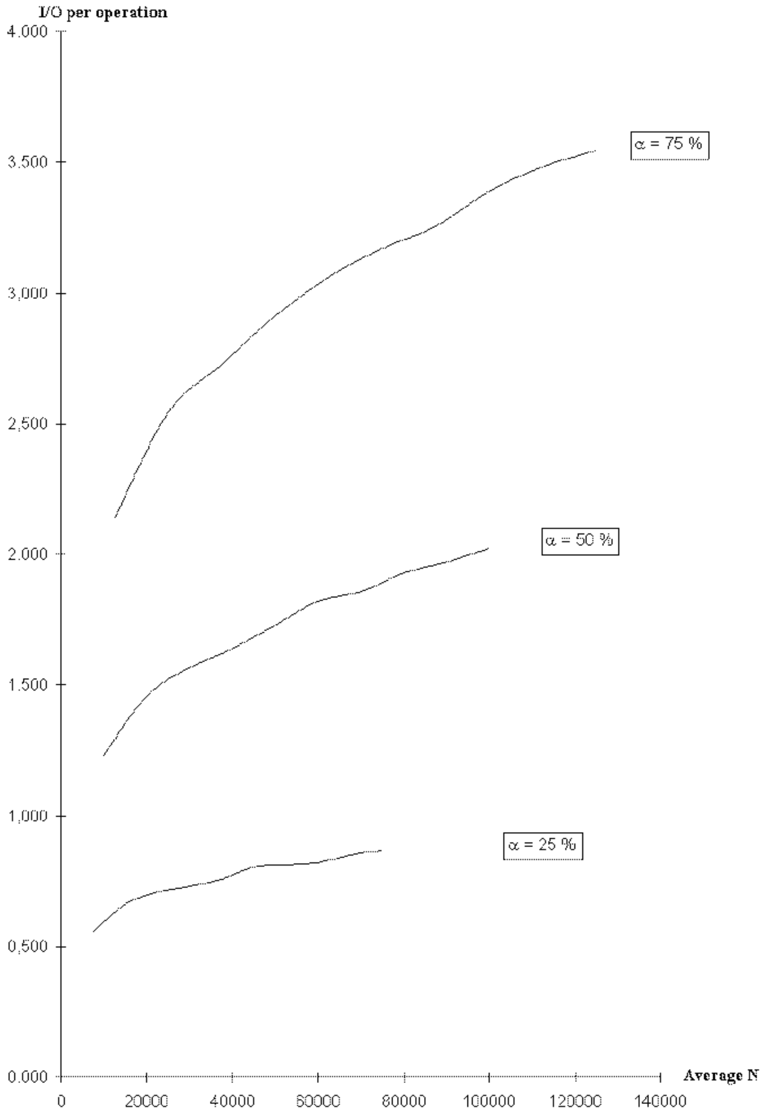

Figure 9.3: B-tree with m=50 and B=50

the complexity for performing delete-max is $O(\frac{1}{B}\log_B n)$, we expect smaller α's will give lower I/O's. The chart also shows this relationship.

Figure 9.4 shows the B-tree priority queue with varying internal memory size. Increased

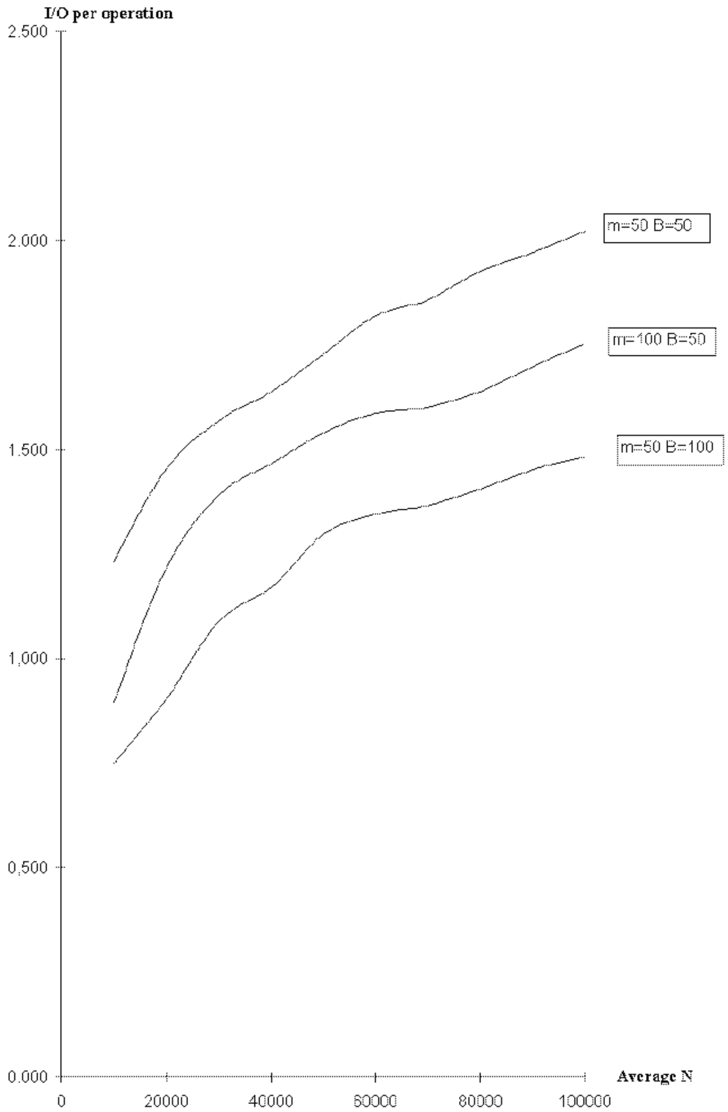

Figure 9.4: B-tree with $\alpha = 50\%$

B clearly shows an improvement in the number of I/O's. This reduction comes from both a bigger fan-out (lower height) and a bigger delete-max buffer.

9.3 I/O-queue

The I/O-queue is implemented as described in Section 4.3. The algortihm works using the external-memory simulator, with the internal memory having equal sized insert and root buffer. The size of the nodes are not M but about $\frac{M}{2}$ (as the internal buffers). The implementations are not agressively optimized, specially the shift-up procedure can be improved. Figure 9.5 shows the I/O-queue with varying α. Since the external complexity

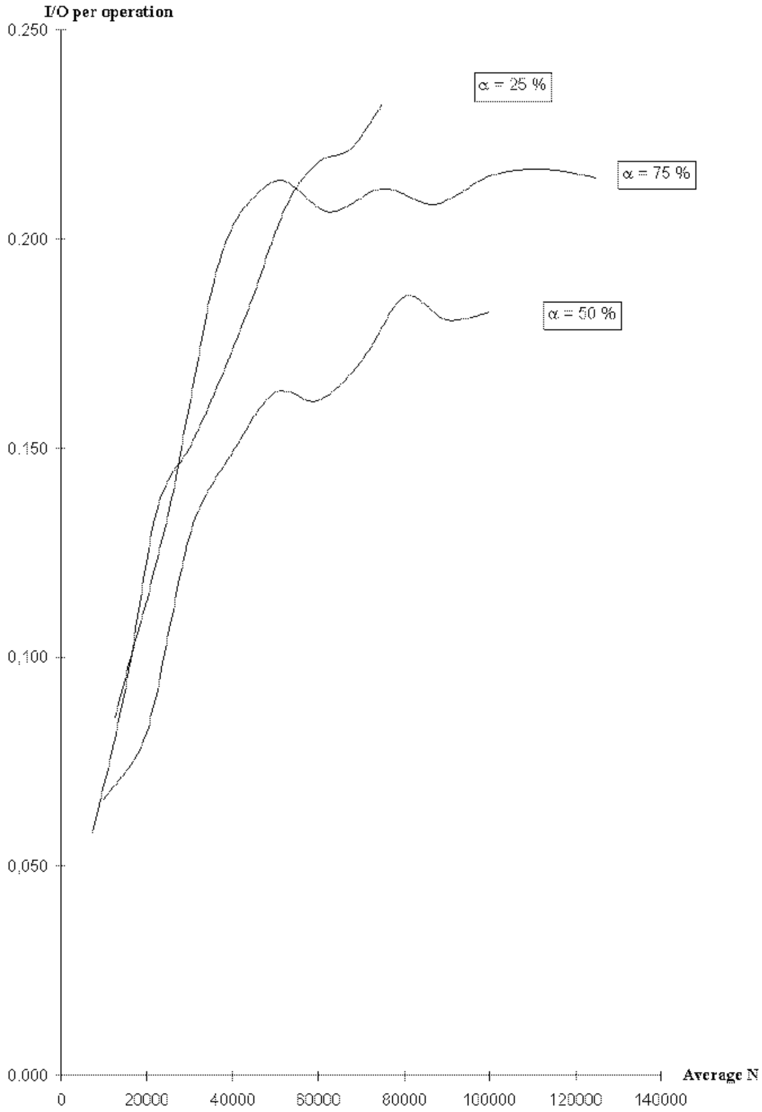

Figure 9.5: I/O-queue with m=50 and B=50

for the two operations both are $O(\frac{1}{B}\log_m n)$, we cannot in advance foretell any difference using different α's. But we suspect that lower α's give a higher number of I/O's, since low α's give many refills (m-way merges). Based on the chart, we can see that many insertions or many delete-max's give higher I/O's. This could be explained by the fact that many insertions cause many new leaves to be created, and many delete-max's cause many leaf melt operations. The case with $\alpha = 50\%$ seems to be a good compromise. To see if this pattern continues, we try the I/O-queue with larger N's, see Figure 9.6. As the chart shows, the pattern seem to continue. This means that the I/O-queue works best when the

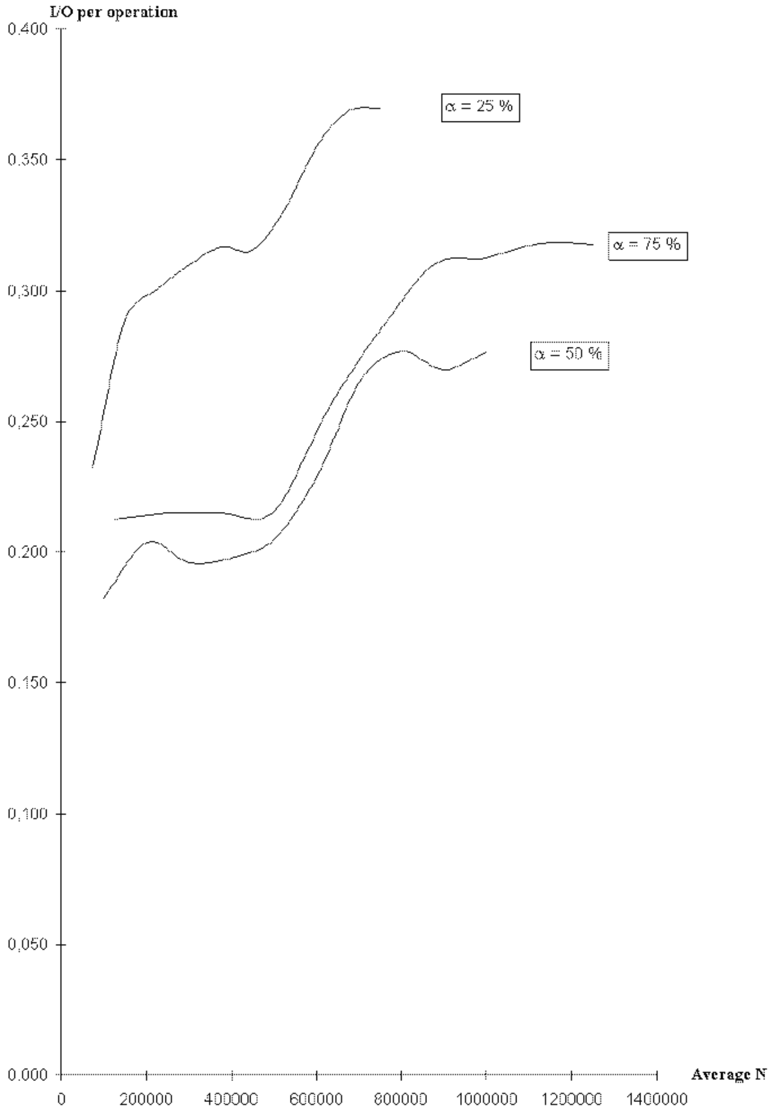

Figure 9.6: I/O-queue with $m = 50$, $B = 50$ and large N's

size of the queue does not change often during the operations.

Figure 9.7 shows the I/O-queue with varying internal memory. As usual, increasing B

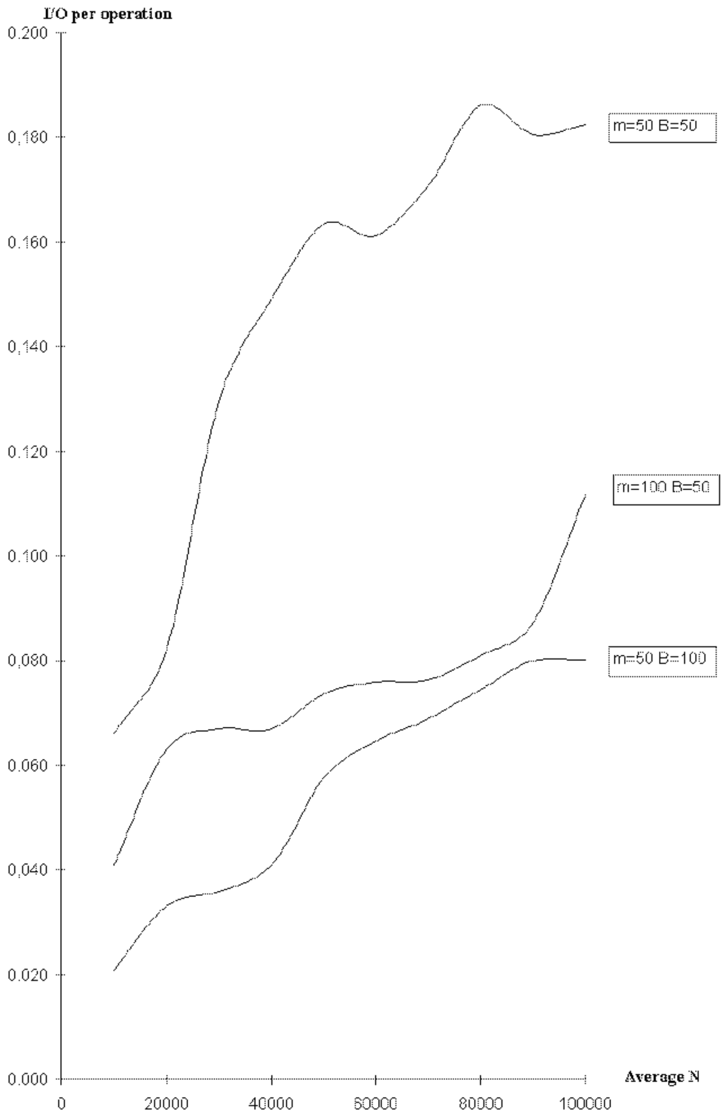

Figure 9.7: I/O-queue with $\alpha = 50\%$

means more than increasing m. The chart also shows that increasing the internal memory has an effect. This is due to the fact that the buffers are increased.

9.4 Comparison of the structures

In the following we compare the queue structures with varying N and α, but fixed B and m's. Figures 9.8, 9.9 and 9.10 show the results.

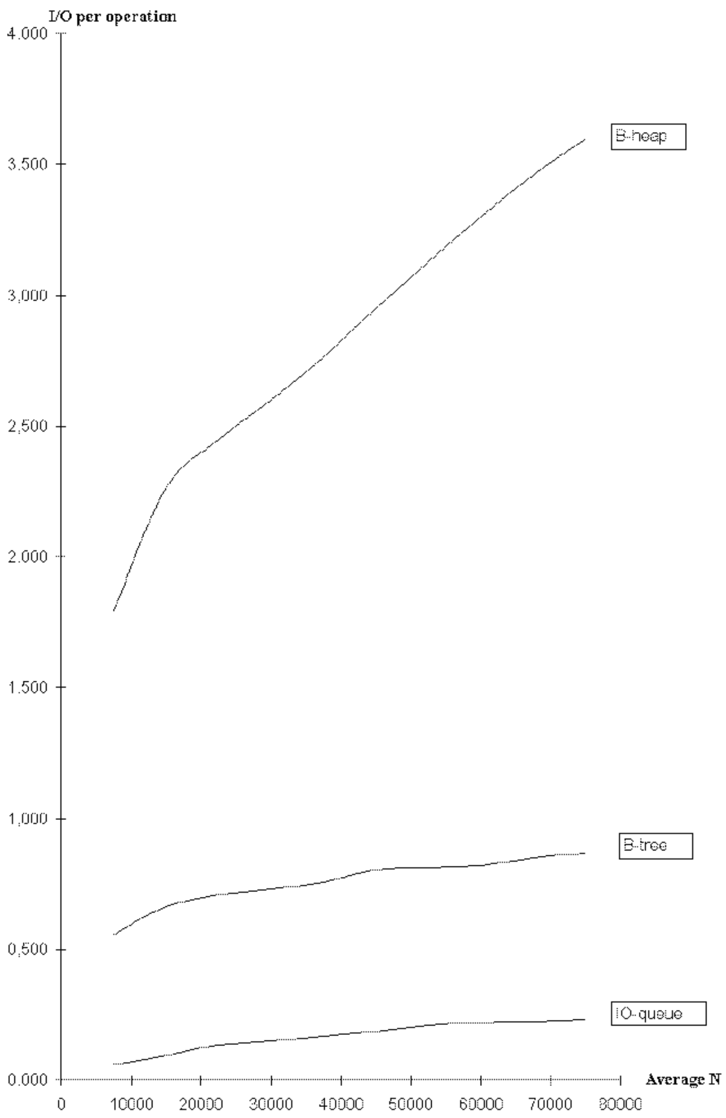

Figure 9.8: The queues with $\alpha = 25\%$, m=50 and B=50

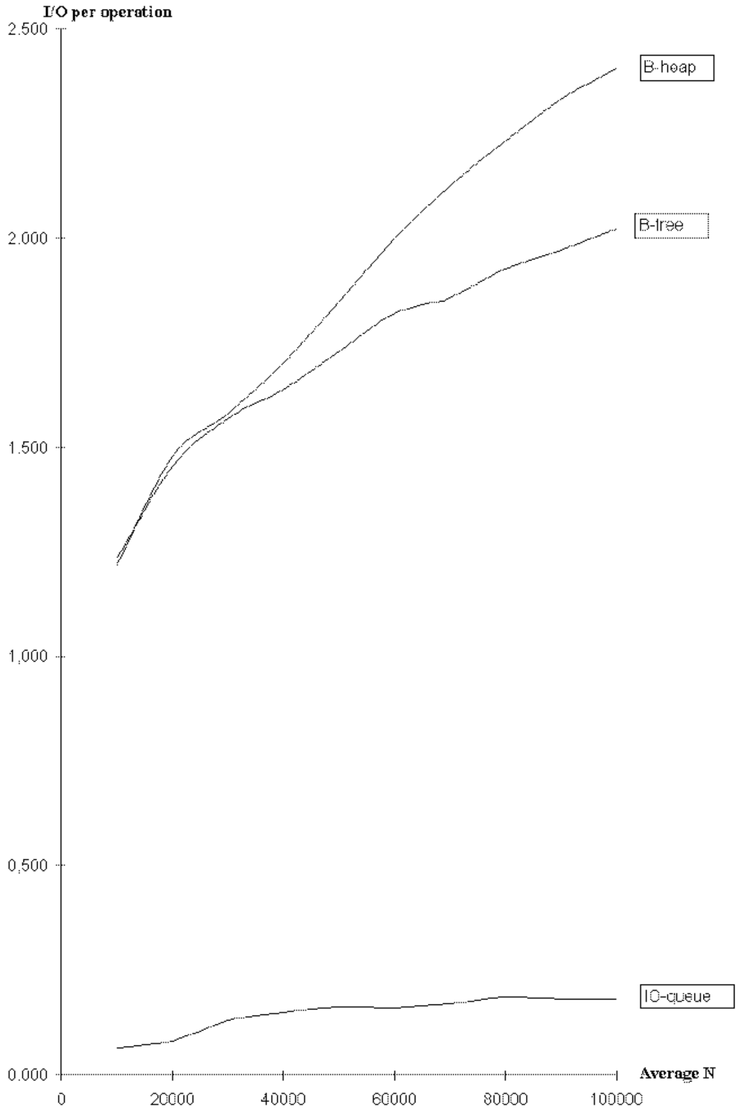

Figure 9.9: The queues with $\alpha = 50\%$, m=50 and B=50

The I/O-queue is clearly the best, and has stable results compared to the other two. As

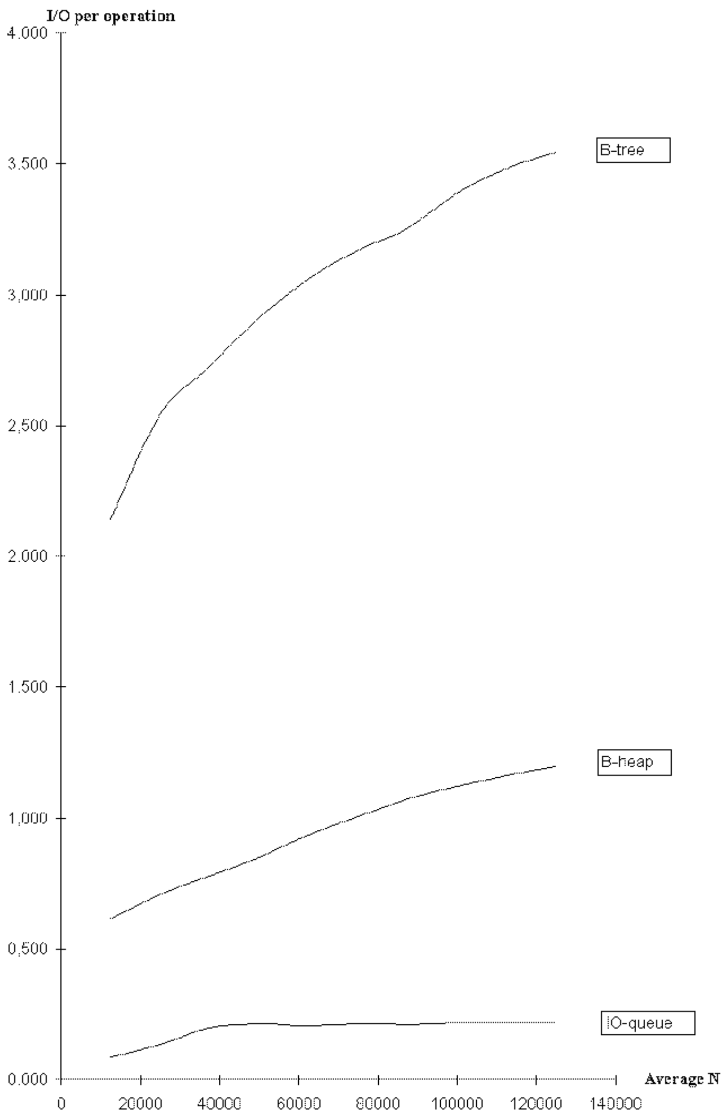

Figure 9.10: The queues with $\alpha = 75\%$, m=50 and B=50

seen B-heaps and B-trees compete to be the second best. When the number of insertions are biggest, $\alpha = 75\%$, the B-heap is the better, and when $\alpha = 25\%$ the B-trees is the better.

9.5 Conclusion

The performed test shows that the I/O-queue is the best choice, but testing the structures in another way might have given a different result. If for example the operations were generated so that the sizes of the structures changed often, the I/O-queue would not work so exceptionally well. The tests also show that the I/O-queue is good in pratice, not just in theory.

In order to choose between the B-tree and the B-heap, one would have to examine the algorithm in which it should be used. Many insertions favour the B-heap, and many delete-max's favour the B-tree.

Chapter 10

Conclusion

We defined a memory model which is relatively simple and hopefully realistic enough to reflect the behaviour of real computers. The model is used to analyse the time complexity of different algorithms.

10.1 Results

One of the main results from this thesis is that efficient internal algorithms do not necessarily have efficient external complexities. The external complexity can in many problems be improved wihtout the loss of internal complexity. The external complexity of the dictionaries operations are improved from $\log_2 N$ (AVL-trees) to $\log_B n$ (B-trees). Hashing results are also improved by extending the methods used. We introduced a new priority queue structure (the *I/O-queue* where the insert and delete-max operations both have the amortized external complexity of $O(\frac{1}{B} \log_m n)$. This is a big improvement compared to B-heaps and B-trees. When supporting the increase-key operation the complexities deteriorate. This is true for all the priority queues examined. It is relativly easy to reach the external complexity lower bound for sorting ($\emptyset(nlog_m n)$). Mergesort, samplesort and heapsort variants match this lower bound. When dealing with graph problems, I/O-efficient simulation of PRAM algorithms gives good results.

In the second part of the paper we simulated the behaviour of a two-level memory executing sorting and priority queue algorithms. The results in general matched the results of the theoretical analysis. The external algorithm implementing m-way mergesort appeared to be the most efficient sorting algorithm. Tests of the priority queues showed that the I/O-queue was clearly the most I/O-efficient.

10.2 I/O-efficient algorithms

The main difference between an internal algorithm and an I/O-efficient external algorithm is that the internal algorithms see the memory as an array of records, while external algorithms recognize the blocking of records. For example extendible hashing (see Section 3.4.4) maps the key-values to B-blocks in external memory in contrary to hashing with chaining (see Section 3.4.2) where key-values are mapped into array of records. In I/O-queues the blocks handled are of size m B-blocks. The m-blocks are created using an internal buffer during inserting, and records are always handled within these blocks.

Another way to make I/O-efficient algorithms is to use all m internal frames while handling blocks, that is: when B-blocks are processed, no frames in internal memory should contain unnecessary B-blocks. For example the external complexity of m-way merging and 2-way mergings are the same when the number of records to merge are the same, but more lists are merged with m-way merging.

Since external algorithms do not rely on a page replacement strategy one can always administer the page replacement efficiently. For example: an external algorithm implementing a search tree will keep the top nodes in internal memory since they are used often.

10.3 Further work

Our analysis and simulation are performed within the defined model. It would be interesting to examine how realistic the model is, that is executing the algorithms on a real computer and measuring real time. Specially an evalution of the I/O-queue versus other I/O-efficient priority queues. An implementation of Buffer trees (see Section 3.3) would decide if the Buffer tree is as efficient as the I/O-queue.

Other suggestions are:

- Further work on and implemention of increase-key operations on the priority queues.
- Analysing and designing adaptive sorting algorithms with respect to external complexity.
- Are there $O(e \log_m e)$ graph algorithms that solve non-triviel problems.
- As with adaptive sorting algorithms, can one make graph algorithms that have upper bounds depending on the pattern of input. For example: can one make a locality measure that is a factor in the external complexity of a graph searching algorithm.
- Designing a priority queue as efficient as the I/O-queue but with worst-case complexitites instead of amortized complexities.
- Analysing space requirements of the different algorithms.
- Refining the model making it more realistic.

Part III

Appendices

Appendix A

Small mathematical propositions

Proposition A.1 *Let $T \geq 1$ be an integer. The function $F(T) = T \log_T N$ has a minimum at $T = 3$.*

Proof $F(T)$

$$= T \log_T N$$

$$= T \frac{\ln N}{\ln T}$$

This means that

$$F'(T)$$

$$= \ln N \frac{\ln T - T \frac{1}{T}}{(\ln T)^2}$$

$$= \ln N \frac{\ln T - 1}{(\ln T)^2}$$

Which gives us an extremum in $T = e$. Since $F'(T) < 0$ when $T < e$, and $F'(T) > 0$ for $T > e$, the function $F(T)$ has a minimum in $T = e$. Since $F(2) = 2 \log_2 N$ is larger than $F(3) = 3 \log_3 N = \frac{3}{\log_2 3} \log_2 N$, the function $F(T)$, has a minimum at $T = 3$, when T is an integer. □

Proposition A.2 *Let $T \geq 1$ and $B \geq 3$ be integers. The function $F(T) = \lceil \frac{T}{B} \rceil \log_T N$ has a minimum at $T = B$.*

Proof If $k \geq 1$ is an integer, there is no point in checking values other than $T = kB$, since $\lceil \frac{T}{B} \rceil$ does not increase when T is increased to the nearest kB value, but $\log_T N$ decreases. We get that

$$F(kB)$$

$$= \lceil \frac{kB}{B} \rceil \log_{kB} N$$

$$= k \log_{kB} N$$

$$= \frac{k}{\log_B kB} \log_B N$$

$$= \frac{k}{\log_B k+1} \log_B N.$$

Since $B \geq 3$, this function is increasing in k, giving us a minimum at $k = 1$, or $T = B$. □

Proposition A.3 $\sum_{i=0}^{k} r^i = \frac{r^{k+1}-1}{r-1}$.

Proof From [38] eq. 19.4, we have

$$a + ar^2 + ar^3 + \cdots + ar^{k-1} = \frac{a(1-r^k)}{1-r}$$

$$\Leftrightarrow \sum_{i=0}^{k-1} ar^i = \frac{a(1-r^k)}{1-r}$$

When $a = 1$, we get

$$\sum_{i=0}^{k-1} r^i = \frac{1-r^k}{1-r}$$

$$\Leftrightarrow \sum_{i=0}^{k} r^i = \frac{1-r^{k+1}}{1-r}$$

$$\Leftrightarrow \sum_{i=0}^{k} r^i = \frac{r^{k+1}-1}{r-1} \quad \square$$

Appendix B

The two simulators

VirCell.h Standard declarations such as B and m, and the record type.

VirMem.h The virtual memory simaltor.

ExtMem.h The external memory simulator.

VirCell.h : Standard declarations

```
/*****************************************************************/
/*  The class VirCell is used in VirMem and ExtMem              */
/*****************************************************************/
#ifndef VirCellCode
#define VirCellCode

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

#define BlockSize 100          /* # of elements in a block       */
#define NrOfBlocksInMem 100    /* # of blocks in internal memory */
#define TYPE long


/* Resource measuring */
long MemAcces;
long MemComp;
long MemWrite;
long MemRead;
long BlockReads;
long BlockWrites;


class VirCell {
  private:
    TYPE  *Value;
  public:
    VirCell(TYPE *aValue);
    ~VirCell();

    VirCell operator=(VirCell VC);
    VirCell operator=(TYPE T);
    operator TYPE();
    operator TYPE *();

    int operator==(VirCell VC);
```

```
    int operator==(TYPE T);

    int operator<(VirCell VC);
    int operator<=(VirCell VC);
    int operator>(VirCell VC);
    int operator>=(VirCell VC);

    int operator<(TYPE T);
    int operator<=(TYPE T);
    int operator>(TYPE T);
    int operator>=(TYPE T);

    VirCell operator++();
    VirCell operator--();
};

void WriteStatusFile(char* s,long N);
void randomize();
long random(long N);

/*****************************************************************/
/***        WriteStatusFile  and random                        ***/
/*****************************************************************/

void WriteStatusFile(char* s,long N) {
  FILE *out;
  out = fopen(s,"a+");
  fprintf(out,"%ld : %ld : %ld : %ld : %ld : %ld : %ld : %ld : %ld\n",
                                  N,(long)NrOfBlocksInMem, (long)BlockSize,
                                  MemAcces, MemComp, MemWrite, MemRead,
                                  BlockReads, BlockWrites);
  fclose(out);
}

void randomize() {
  srand48(1);
}

long random(long N) {
  return (long)(N*drand48());
}

/*****************************************************/
/***                  VirCell                      ***/
/*****************************************************/

VirCell::VirCell(TYPE *aValue) {
  Value = aValue;
}


VirCell::~VirCell() {
}

VirCell VirCell::operator=(VirCell VC) {
  MemWrite++;
  MemRead++;
  *(*this).Value = *VC.Value;
  return *this;
}
```

```
VirCell VirCell::operator=(TYPE T) {
  MemWrite++;
  *(*this).Value = T;
  return *this;
}

VirCell::operator TYPE() {
  MemRead++;
  return *(*this).Value;
}

VirCell::operator TYPE *() {
  MemRead++;
  return (*this).Value;
}

int VirCell::operator==(VirCell VC) {
  MemComp++;
  MemRead += 2;
  return (*(*this).Value == *(VC.Value));
}

int VirCell::operator==(TYPE T) {
  MemComp++;
  MemRead++;
  return (*(*this).Value == T);
}

int VirCell::operator<(VirCell VC) {
  MemComp++;
  MemRead += 2;
  return (*(*this).Value < *(VC.Value));
}

int VirCell::operator<=(VirCell VC) {
  MemComp++;
  MemRead += 2;
  return (*(*this).Value <= *(VC.Value));
}

int VirCell::operator>(VirCell VC) {
  MemComp++;
  MemRead += 2;
  return (*(*this).Value > *(VC.Value));
}

int VirCell::operator>=(VirCell VC) {
  MemComp++;
  MemRead += 2;
  return (*(*this).Value >= *(VC.Value));
}


int VirCell::operator<(TYPE T) {
  MemComp++;
  MemRead++;
  return (*(*this).Value < T);
}

int VirCell::operator<=(TYPE T) {
  MemComp++;
  MemRead++;
```

```
  return (*(*this).Value <= T);
}

int VirCell::operator>(TYPE T) {
  MemComp++;
  MemRead++;
  return (*(*this).Value > T);
}

int VirCell::operator>=(TYPE T) {
  MemComp++;
  MemRead++;
  return (*(*this).Value >= T);
}

VirCell VirCell::operator++() {
  MemWrite++;
  (*(*this).Value)++;
  return *this;
}

VirCell VirCell::operator--() {
  MemWrite++;
  (*(*this).Value)--;
  return *this;
}


#endif
```

VirMem.h : Virtual memory simulator

```
/******************************************************************/
/*  The class VirMem simulates an virtual LRU memory             */
/******************************************************************/
#ifndef VirMemCode
#define VirMemCode

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "VirCell.h"


class VirMem {
  private:
    TYPE *Mem;
    long MemSize;
    long NrOfBlocks;                    /* # of blocks in all */

    long BlocksInMem[NrOfBlocksInMem];  /* The blocks in internal memory */

    int BlockInMem(long Block);         /* Are Block in internal memory */
    void FrontBlock(long Block);        /* Moves block first in LRU-queue */
    void GetBlock(long BLock);          /* Gets Block from external memory */
  public:
    VirMem(long Size);
    ~VirMem();

    void Init();
    VirCell operator[](unsigned long t);
    void Flush();                       /* Writes alle blocks from internal to external memory */
```

```
  void WriteStatus();

};


/****************************************************/
/***                 VirMem                       ***/
/****************************************************/

VirMem::VirMem(long Size) {
  MemSize = Size;
  Mem = (TYPE *) calloc(MemSize,sizeof(TYPE));
  if (Mem == NULL) {
    printf("Not enough free memory to Mem \n");
    exit(1);
    }
  NrOfBlocks = MemSize/BlockSize + 1;
  Init();
}


VirMem::~VirMem() {
  free(Mem);
}

void VirMem::Init() {
  long a;
  for(a = 0; a < NrOfBlocksInMem; a++)
    BlocksInMem[a]=-1;
  MemAcces = 0;
  MemComp = 0;
  MemWrite = 0;
  MemRead = 0;
  BlockReads = 0;
  BlockWrites = 0;
}

int VirMem::BlockInMem(long Block) {

  int temp = 0;
  for(int a = 0; a < NrOfBlocksInMem; a++)
    if (BlocksInMem[a] == Block)
      temp = 1;
  return temp;
}

void VirMem::FrontBlock(long Block) {
  /* Find old place */
  int Place;
  for(Place = 0; BlocksInMem[Place] != Block; Place++);
  /* Insert block */
 for(int a = Place; a > 0; a--)
   BlocksInMem[a] = BlocksInMem[a-1];
 BlocksInMem[0] = Block;
}

void VirMem::GetBlock(long Block) {
  BlockReads++;
  /* remove last block */
  long LastBlock = BlocksInMem[NrOfBlocksInMem-1];
  if (LastBlock != -1)
```

```
    BlockWrites++;
  /* Insert new block */
  for(int a = NrOfBlocksInMem-1; a > 0; a--)
    BlocksInMem[a] = BlocksInMem[a-1];
  BlocksInMem[0] = Block;
}


VirCell VirMem::operator[](unsigned long t) {
  if (t < MemSize) {
    int a,b;

    VirCell VC((TYPE*) &Mem[t]);
    long Block = t/BlockSize;
    MemAcces++;
    if (BlockInMem(Block))
      FrontBlock(Block);
    else
      GetBlock(Block);
    return VC;
    }
  else {
    printf("Adress %ld is not in mem\n",(long) t);
    exit(0);
    return NULL;
    }
}

void VirMem::Flush() {
  for(int a = 0; a < NrOfBlocksInMem; a++) {
    if (BlocksInMem[a] != -1)
      BlockWrites++;
    BlocksInMem[a] = -1;
    }
}

void VirMem::WriteStatus() {
  for(int a=0; a < NrOfBlocksInMem; a++)
    if (BlocksInMem[a] != -1) {
      printf("Block nr. %ld is still in internal memory\n",a);
      exit(42);
      }
  printf("Nr. of memory acceses : %ld\n",MemAcces);
  printf("Nr. of comparisons    : %ld\n",MemComp);
  printf("Nr. of writes         : %ld\n",MemWrite);
  printf("Nr. of reads          : %ld\n",MemRead);
  printf("Nr. of block reads    : %ld\n",BlockReads);
  printf("Nr. of block writes   : %ld\n",BlockWrites);
}

#endif
```

ExtMem.h : External memory simulator

```
/****************************************************************/
/*  The class ExtMem simulates an external memory.              */
/****************************************************************/
#ifndef ExtMemCode
#define ExtMemCode

#include <malloc.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include "VirCell.h"


class ExtMem {
  private:
    TYPE  *InternMem;
    TYPE  *ExternMem;
    long InternMemSize;
    long NrOfBlocks;                    /* # of blocks in external memory */

  public:
    ExtMem(long Size);
    ~ExtMem();

    void Init();
    VirCell operator[](unsigned long t);

    void BlockRead(long ExtBlock,long IntBlock);
    void BlockWrite(long IntBlock, long ExtBlock);

    void WriteStatus();

    /* Functions to help test a program */
    TYPE ReadExt(long ExtAdr);
    void WriteExt(long ExtAdr, TYPE Element);
};


/*********************************************************/
/***                    ExtMem                       ***/
/*********************************************************/

ExtMem::ExtMem(long Size) {
  InternMemSize = NrOfBlocksInMem*BlockSize;
  NrOfBlocks = Size/BlockSize + 1;

  ExternMem = (TYPE *) calloc(Size,sizeof(TYPE));
  if (ExternMem == NULL) {
    printf("Not enough free memory to ExternMem \n");
    exit(1);
    }
  InternMem = (TYPE *) calloc(InternMemSize,sizeof(TYPE));
  if (InternMem == NULL) {
    printf("Not enough free memory to InternMem \n");
    exit(1);
    }
  Init();
}


ExtMem::~ExtMem() {
  free(InternMem);
  free(ExternMem);
}

void ExtMem::Init() {
  MemAcces = 0;
  MemComp = 0;
  MemWrite = 0;
  MemRead = 0;
  BlockReads = 0;
```

```
  BlockWrites = 0;
}

VirCell ExtMem::operator[](unsigned long t) {
  if (t < InternMemSize) {
    VirCell VC((TYPE*) &InternMem[t]);
    MemAcces++;
    return VC;
    }
  else {
    printf("Adress %ld is not in internal mem\n",(long) t);
    exit(0);
    return NULL;
    }
}

void ExtMem::BlockRead(long ExtBlock, long IntBlock) {
  if ((ExtBlock < NrOfBlocks) && (IntBlock < NrOfBlocksInMem)) {
    BlockReads++;
    for(int a=0; a < BlockSize; a++)
      InternMem[IntBlock*BlockSize+a] = ExternMem[ExtBlock*BlockSize+a];
    }
  else {
    printf("BlockRead(%ld,%ld) is not valid\n",ExtBlock,IntBlock);
    exit(0);
    }
}


void ExtMem::BlockWrite(long IntBlock, long ExtBlock) {
  if ((ExtBlock < NrOfBlocks) && (IntBlock < NrOfBlocksInMem)) {
    BlockWrites++;
    for(int a=0; a < BlockSize; a++)
      ExternMem[ExtBlock*BlockSize+a] = InternMem[IntBlock*BlockSize+a];
    }
  else {
    printf("BlockWrite(%ld,%ld) is not valid\n",IntBlock,ExtBlock);
    exit(0);
    }
}


void ExtMem::WriteStatus() {
  printf("Nr. of memory acceses : %ld\n",MemAcces);
  printf("Nr. of comparisons    : %ld\n",MemComp);
  printf("Nr. of writes         : %ld\n",MemWrite);
  printf("Nr. of reads          : %ld\n",MemRead);
  printf("Nr. of block reads    : %ld\n",BlockReads);
  printf("Nr. of block writes   : %ld\n",BlockWrites);
}

TYPE ExtMem::ReadExt(long ExtAdr) {
  return ExternMem[ExtAdr];
}

void ExtMem::WriteExt(long ExtAdr, TYPE Element) {
  ExternMem[ExtAdr] = Element;
}


#endif
```

Appendix C

The sorting programs

sort.c Internal version of samplesort and mergesort.

sortext.c External versions of samplesort and mergesort.

Btree-virtual.c Virtual memory version of B-tree heapsort.

Btree-external.c External version of B-tree heapsort.

heapsort.c Binary heapsort and I/O-queue heapsort.

sort.c : Internal version of samplesort and mergesort

```
/***********************************************************/
/***  Test af Merge og Quicksort, samt nye versioner      ***/
/***********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "VirMem.h"

#define MaxSize 100000
#define n N/BlockSize     /* We assume n is an integer */
#define m NrOfBlocksInMem

#define min(a,b) (a < b) ? a : b

VirMem VMem(2*MaxSize);

TYPE Data[MaxSize];
long N;

void SwapData(long t1, long t2) {
  TYPE t = Data[t1];
  Data[t1] = Data[t2];
  Data[t2] = t;
}

void MakeNewData() {
  long a;
  TYPE t;
  randomize;
  for(a = 0; a < N; a++)
    Data[a] = a;
  for(a = 0; a < N; a++)
    SwapData(random(N),random(N));
}
```

```
void VMakeData() {
  long a;
  for(a=0; a < N; a++)
    VMem[a] = Data[a];
}

int VDataSorted() {
  int Temp;
  long a;

  Temp = 1;
  for(a=0; a < N; a++)
    if (a != (long) VMem[a] )
      Temp = 0;
  return Temp;
}


/************            Quicksort                  ***********/

void Vswap(long i, long j) {
  TYPE Temp;

  Temp = VMem[i];
  VMem[i] = VMem[j];
  VMem[j] = Temp;
}

void QuickSort(long begin, long end) {
  long i,j;
  TYPE Piv,Temp;
  if (end - begin > 0) {
    Piv = VMem[begin];
    j = end;
    i = begin+1;
    while (i < j) {
      if (VMem[i] > Piv) {
        while ((VMem[j] > Piv) && (i < j))
          j--;
        if (i < j)
          Vswap(i,j);
        }
      else
        i++;
      }
    if (VMem[i] <= Piv)
      Vswap(begin,i);
    else {
      i--;
      Vswap(begin,i);
      }
    QuickSort(begin,i-1);
    QuickSort(i+1,end);
    }
}

/************            Mergesort                  ***********/

void MergeSort(long begin, long end) {
  long mid;
  long a,b,c;
```

```
  if (end > begin) {
    mid = (begin+end)/2;
    MergeSort(begin,mid);
    MergeSort(mid+1,end);

    /* Copy first list to end of data, and merge them */
    for(a=0; a <= mid-begin; a++)
      VMem[N+a] = VMem[a+begin];
    b = mid+1;
    c = N;
    for(a=begin; a <= end; a++) {
      if (((b <= end) && (VMem[b] <= VMem[c])) || (c > N+mid-begin)) {
        VMem[a] = VMem[b];
        b++;
        }
      else {
        VMem[a] = VMem[c];
        c++;
        }
      }
    }
}



/******                MAIN                        *****/

void RUN(long t) {
  N = t;

  printf("*****************************\n");
  printf("Sorting %d elements\n\n",N);
  MakeNewData();

  printf("Quicksort :\n");
  VMakeData();
  VMem.Init();
  QuickSort(0,N-1);
  VMem.Flush();
  VMem.WriteStatus();
  WriteStatusFile("quickVir.out",N);

  printf("Mergesort :\n");
  VMakeData();
  VMem.Init();
  MergeSort(0,N-1);
  VMem.Flush();
  VMem.WriteStatus();
  WriteStatusFile("mergeVir.out",N);

}

void main () {

        RUN(100);
        RUN(500);
        RUN(1000);
        RUN(5000);
        for(int qq = 10000; qq <= 100000; qq += 10000)
          RUN(qq);
  printf("Program end\n");
```

```
}
```

sortext.c : External version of samplesort and mergesort

```
/**********************************************************/
/***  Test af Merge og Quicksort, samt nye versioner      ***/
/**********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "ExtMem.h"

#define MaxSize 100000
#define n N/BlockSize     /* We assume n is an integer */
#define m NrOfBlocksInMem

#define min(a,b) (a < b) ? a : b

ExtMem EMem((long) (m-2)*MaxSize);
long Pivot[m-3];
long Counter[m-2];    // used in ext mergesort

TYPE Data[MaxSize];
long N;

void SwapData(long t1, long t2) {
  TYPE t = Data[t1];
  Data[t1] = Data[t2];
  Data[t2] = t;
}

void MakeNewData() {
  long a;
  TYPE t;
  randomize;
  for(a = 0; a < N; a++)
    Data[a] = a;
  for(a = 0; a < N; a++)
    SwapData(random(N),random(N));
}

void EMakeData() {
  long a;
  for(a=0; a < N; a++)
    EMem.WriteExt(a,Data[a]);
}

int EDataSorted() {
  int Temp;
  long a;

  Temp = 1;
  for(a=0; a < N-1; a++)
    if (EMem.ReadExt(a) > EMem.ReadExt(a+1))
      Temp = 0;
  return Temp;
}


/******************************************************************/
/************            Extended Quicksort            ***********/
/******************************************************************/
```

```
void SortPivot() {
  for(int a = 0; a < m-4; a++)
    for(int b = a+1; b < m-3; b++) {
      if (Pivot[a] > Pivot[b]) {
        long T = Pivot[a];
        Pivot[a] = Pivot[b];
        Pivot[b] = T;
        }
      }
}

void Eswap(long i, long j) {

  TYPE Temp;

  Temp = EMem[i];
  EMem[i] = EMem[j];
  EMem[j] = Temp;
}

void RecurseQuickSort(long begin, long end) {
  long i,j;
  TYPE Piv,Temp;
  if (end - begin > 0) {
    Piv = EMem[begin];
    j = end;
    i = begin+1;
    while (i < j) {
      if (EMem[i] > Piv) {
        while ((EMem[j] > Piv) && (i < j))
          j--;
        if (i < j)
          Eswap(i,j);
        }
      else
        i++;
      }
    if (EMem[i] <= Piv)
      Eswap(begin,i);
    else {
      i--;
      Eswap(begin,i);
      }
    RecurseQuickSort(begin,i-1);
    RecurseQuickSort(i+1,end);
    }
}

void NormalQuickSort(long begin, long end) {
  if (end > begin) {
    int a;
    for(a = 0; a <= end/BlockSize - begin/BlockSize; a++)
      EMem.BlockRead(a+begin/BlockSize,a);
    RecurseQuickSort(begin % BlockSize,begin % BlockSize + end-begin);
    for(a = 0; a <= end/BlockSize - begin/BlockSize; a++)
      EMem.BlockWrite(a,a+begin/BlockSize);
    }
}

int GetOutputMemAdr(int Nr) {
  int temp = Nr*BlockSize+Counter[Nr] % BlockSize;
  return temp;
```

```
}

int GetOutputBlock(int Nr) {
  int temp = n*(Nr+1)+Counter[Nr]/BlockSize-1;
  return temp;
}

void ExtQuickSort(long begin, long end) {
  if (end/BlockSize-begin/BlockSize < m)
    NormalQuickSort(begin,end);
  else {
    long a;
    int b,NewList;
    /* Select and sort pivot elements */
    for(b = 0; b < m-4; b++) {
      if (((b+begin % BlockSize) % BlockSize == 0) || (b==0))
        EMem.BlockRead((begin+b)/BlockSize,m-3);
      Pivot[b] = EMem[(m-3)*BlockSize+(b+begin % BlockSize) % BlockSize];
      }
    SortPivot();
    /* Init counters */
    for(b = 0; b < m-3; b++)
      Counter[b] = 0;
    for(a = begin; a <= end; a++) {
      /* Check if input block is in memory */
      if ((a % BlockSize == 0) || (a == begin))
        EMem.BlockRead(a/BlockSize,m-3);
      /* Find new list number */
      TYPE t = EMem[(m-3)*BlockSize + a % BlockSize];
      NewList = 0;
      for(b = 0; b < m-4; b++)
        if (Pivot[b] < t)
          NewList = b+1;
      /* Move element and update counters */
      EMem[GetOutputMemAdr(NewList)] = t;
      Counter[NewList]++;
      /* Write output block */
      if (Counter[NewList] % BlockSize == 0)
        EMem.BlockWrite(NewList,GetOutputBlock(NewList));
      }
    /* Write last list */
    for (a = 0; a < m-3; a++)
      EMem.BlockWrite(a,GetOutputBlock(a)+1);
    /* Copy elements back */
    a = begin;
    if (begin % BlockSize != 0)
      EMem.BlockRead(begin/BlockSize,0);
    for(b = 0; b < m-3; b++) {
      for(long c = 0; Counter[b] > c; c++) {
        if (c % BlockSize == 0)
          EMem.BlockRead(n+n*b+c/BlockSize,1);
        if ((a % BlockSize == 0) && (a/BlockSize == end/BlockSize))
          EMem.BlockRead(a/BlockSize,0);
        EMem[a % BlockSize] = EMem[BlockSize+c % BlockSize];
        if ((a % BlockSize == BlockSize - 1) || (a == end))
          EMem.BlockWrite(0,a/BlockSize);
        a++;
        }
      }
    /* Call recursively */
    long ListSplit[m-2];
    ListSplit[0] = begin;
```

```
    for(a = 0; a < m-4; a++)
      ListSplit[a+1] = ListSplit[a] +(long) Counter[a];
    ListSplit[m-3] = end+1;
    for(a = 0; a < m-3; a++)
      ExtQuickSort(ListSplit[a],ListSplit[a+1]-1);
    }
}

/*************************************************************/
/************            Ext MergeSort            ***********/
/*************************************************************/

void NormalMergeSort(long begin, long end) {
  long mid;
  long a,b,c;
  long Free = 2*m/3*BlockSize;

  if (end > N-1)
    end = N-1;

  if (end > begin) {
    mid = (begin+end)/2;
    NormalMergeSort(begin,mid);
    NormalMergeSort(mid+1,end);

    /* Copy first list to end of data, and merge them */
    for(a=0; a <= mid-begin; a++)
      EMem[Free+a] = EMem[a+begin];
    b = mid+1;
    c = Free;
    for(a=begin; a <= end; a++) {
      if ((c > Free+mid-begin)  || ((b <= end) && (EMem[b] <= EMem[c])))
        EMem[a] = EMem[b++];
      else
        EMem[a] = EMem[c++];
      }
    }
}

int GetListMemAdr(int Nr) {
  int temp = Nr*BlockSize+Counter[Nr] % BlockSize;
  if (temp < 0)
    temp = 0;
  return temp;
}


void MakeMerge(long begin, long size) {
  long a,b;
  int ReadInputBlock[m-2];
  int MinList;
  if (begin + size > n-1)
    size = n-begin-1;
  /* Reset counters */
  for(a = 0; a < m-2; a++) {
    Counter[a] = 0;
    ReadInputBlock[a] = 1;
    }
  /* Start merge */
  for(a = 0; (a < size*(m-2)*BlockSize) && (a < N); a++) {
    /* Check if input blocks are i memory */
    for(b = 0; b < m-2; b++) {
```

```
      if (((Counter[b] % BlockSize) == 0) && (ReadInputBlock[b])) {
        EMem.BlockRead(begin+size*b + Counter[b]/BlockSize,b);
        ReadInputBlock[b] = 0;
        }
      if (Counter[b] % BlockSize == 1)
        ReadInputBlock[b] = 1;
      }
    /* Find the minimum element and update the counter */
    MinList = -1;
    for (b = 0; b < m-2; b++)
      if (((MinList == -1) || (EMem[GetListMemAdr(MinList)] > EMem[GetListMemAdr(b)])) &&
          (Counter[b] < size*BlockSize) &&
          (begin+size*b + Counter[b]/BlockSize < n))
        MinList = b;
    EMem[(m-2)*BlockSize + a % BlockSize] = EMem[GetListMemAdr(MinList)];
    ++Counter[MinList];
    /* Write output to disk */
    if (a % BlockSize == BlockSize-1)
      EMem.BlockWrite(m-2,a/BlockSize + n);
    }
  /* Copy input back in place */
  long HuskI = BlockReads;
  long HuskO = BlockWrites;
  TYPE temp = 0;
  for(a = 0; (a < size*(m-2)) && (a+begin < n); a++) {
    EMem.BlockRead(a+n,0);
    for(b = 0; b < BlockSize; b++) {
      if (EMem[b] < temp) {
        printf("Fejl : begin = %ld   size = %ld\n",(long)begin,(long)size);
        }
      temp = EMem[b];
      }
    EMem.BlockWrite(0,a+begin);
    }
  BlockReads = HuskI;
  BlockWrites = HuskO;
}

void ExtMergeSort() {
  int a;
  /* Sort blocks of 2/3 m size */
  long block;
  for(block = 0; block < n; block+=2*m/3) {
    for(a = block; a < block+2*m/3; a++)
      EMem.BlockRead(a,a-block);
    NormalMergeSort(0,2*m/3*BlockSize-1);
    for(a = block; a < block+2*m/3; a++)
      EMem.BlockWrite(a-block,a);
    }
  /* Merge the blocks  */
  long Size = 2*m/3;
  float t;
  while (Size < n) {
    t = n/(1.0*Size)/(m-2);
    if ((int)t != t)
      t = (int) t + 1;
    for(a = 0; a < t; a++)
       MakeMerge(a*Size*(m-2),Size);
     Size*=m-2;
    }
}
```

```
/****************************************************/
/******                MAIN                    *****/
/****************************************************/

void RUN(long t) {
  N = t;

  printf("*****************************\n");
  printf("Sorting %d elements\n\n",N);
  MakeNewData();
/*
  printf("Extended Quicksort :\n");
  EMakeData();
  EMem.Init();
  ExtQuickSort(0,N-1);
  EMem.WriteStatus();
  WriteStatusFile("quickExt.out",N);
*/
  printf("Extended Mergesort :\n");
  EMakeData();
  EMem.Init();
  ExtMergeSort();
  EMem.WriteStatus();
  WriteStatusFile("mergeExt.out",N);

}

void main () {

        RUN(100);
        RUN(500);
        RUN(1000);
        RUN(5000);
        for(int qq = 10000; qq <= 100000; qq += 10000)
          RUN(qq);

  printf("Program end\n");
}
```

btree-virtual.c : Virtual memory version of B-tree heapsort

```
#include <stdio.h>
#include <stdlib.h>
#include "VirMem.h"

#define MaxSize 100000
#define n N/BlockSize     /* We assume n is an integer */
#define m NrOfBlocksInMem

#define T (BlockSize-1)/2 /* The degree of the B-tree, size of the B-node */
#define NIL -1

#define min(a,b) (a < b) ? a : b

VirMem Mem(10*MaxSize);
long LastNode;
long Root;       // Roots blocknumber
long Check ;
long N;

TYPE Data[MaxSize];
```

```
void SwapData(long t1, long t2) {
  TYPE t = Data[t1];
  Data[t1] = Data[t2];
  Data[t2] = t;
}

void MakeData() {
  long a;
  TYPE t;
  randomize;
  for(a = 0; a < N; a++)
    Data[a] = a;
  for(a = 0; a < N; a++)
    SwapData(random(N),random(N));
}

/*************************************************************/
/****                      Diverse                         ***/
/*************************************************************/

long GetPos(long Adr) {    // Returns the blocknumber
  return Adr;
}

int SizePos(long Adr) {
  return GetPos(Adr)*BlockSize+BlockSize-1;
}

void ClearBNode(long Adr) {
  int a;
  for(a = 0; a < T; a++)
    Mem[GetPos(Adr)*BlockSize+a] = 0;
  for(a = 0; a <=T; a++)
    Mem[GetPos(Adr)*BlockSize+T+a] = NIL;
  Mem[SizePos(Adr)] = 0;
}

void MakeNewBNode(long& Adr) {
  Adr = ++LastNode;
  ClearBNode(LastNode);
}

void MakeNewRoot() {
  Root = ++LastNode;
  ClearBNode(Root);
}

/*************************************************************/
/****                  Init and save                       ***/
/*************************************************************/

void InitBTree() {
  Root = NIL;
  LastNode = -1;
}

/*************************************************************/
/****                      Insert                          ***/
/*************************************************************/

void InsertSorted(long Adr, TYPE X, long NewPtr) {
  int a;
```

```
  for(a = (long)Mem[SizePos(Adr)]-1; (a >= 0) && (Mem[GetPos(Adr)*BlockSize+a] > X); a--) {
    Mem[GetPos(Adr)*BlockSize+a+1] = Mem[GetPos(Adr)*BlockSize+a];
    Mem[GetPos(Adr)*BlockSize+a+2+T] = Mem[GetPos(Adr)*BlockSize+a+T+1];
    }
  a++;
  Mem[GetPos(Adr)*BlockSize+a] = X;
  Mem[GetPos(Adr)*BlockSize+a+T+1] = NewPtr;
  ++Mem[SizePos(Adr)];
}

long FindRightChild(long Adr, TYPE X) {
  long P = 0;
  while ((Mem[GetPos(Adr)*BlockSize+P] < X) && (P < (long) Mem[SizePos(Adr)]))
    P++;
  P = Mem[GetPos(Adr)*BlockSize+T+P];
  return P;
}

void SplitNode(long Adr, long& NewNode, TYPE& SplitE) {
  int a,b;
  MakeNewBNode(NewNode);
  a = T/2+1;
  Mem[SizePos(Adr)] = a-1;
  Mem[SizePos(NewNode)] = T-a;
  b = 0;
  while (a < T) {
    Mem[GetPos(NewNode)*BlockSize+b] = Mem[GetPos(Adr)*BlockSize+a];
    Mem[GetPos(Adr)*BlockSize+a] = 0;
    Mem[GetPos(NewNode)*BlockSize+b+T] = Mem[GetPos(Adr)*BlockSize+a+T];
    Mem[GetPos(Adr)*BlockSize+a+T] = NIL;
    a++; b++;
    }
  Mem[GetPos(NewNode)*BlockSize+b+T] = Mem[GetPos(Adr)*BlockSize+a+T];
  Mem[GetPos(Adr)*BlockSize+a+T] = NIL;

  SplitE = Mem[GetPos(Adr)*BlockSize+T/2];
  Mem[GetPos(Adr)*BlockSize+T/2] = 0;
}

void SplitRoot(long& Adr, TYPE X) {
  long NewNode;
  TYPE SplitE;

  SplitNode(Adr,NewNode,SplitE);
  // Make new root
  MakeNewRoot();
  Mem[GetPos(Root)*BlockSize] = SplitE;
  Mem[GetPos(Root)*BlockSize+T] = Adr;
  Mem[GetPos(Root)*BlockSize+T+1] = NewNode;
  Mem[SizePos(Root)] = 1;
  if (X > SplitE)
    Adr = NewNode;
}

void SplitInternalNode(long& Adr, long Father, TYPE X) {
  long NewNode;
  TYPE SplitE;

  SplitNode(Adr,NewNode,SplitE);
  InsertSorted(Father,SplitE,NewNode);
  if (X > SplitE)
```

```
    Adr = NewNode;
}

void FindNode(long Adr, TYPE X) {
  long Child;

  if ((long)Mem[SizePos(Adr)] == 0)
    InsertSorted(Adr,X,NIL);
  else {
    if ((Adr == Root) && (Mem[SizePos(Adr)] == T))
      SplitRoot(Adr,X);
    Child = FindRightChild(Adr,X);
    if (Child == NIL)
      InsertSorted(Adr,X,NIL);
    else {
      if (Mem[SizePos(Child)] == T)
        SplitInternalNode(Child,Adr,X);
      FindNode(Child,X);
      }
    }
}

void BtreeInsert(TYPE X) {
  if (Root == NIL)
    MakeNewRoot();
  FindNode(Root,X);
}

void InsertData() {
  long a;
  for(a=0; a < N; a++)
    BtreeInsert(Data[a]);
}

/*********************************************************/
/****               Error check                     *****/
/*********************************************************/

void Fejl(char* s) {
  printf(s);
  exit(42);
}


/**********************************************************/
/****                ExctractData                     ***/
/**********************************************************/

void InitQueue() {
  Check = (LastNode+1)*BlockSize;
}

void Write(TYPE t) {
  Mem[Check] = t;
  //printf("%ld   ",(long)t);
  if (t != Check-(LastNode+1)*BlockSize)
    Fejl("Wrong number\n");
  Check++;
}

void BTreeExctract(long Adr) {
  for(int a = 0; a <= T; a++) {
```

```
    if ((long)Mem[Adr*BlockSize+a+T] != NIL)
      BTreeExctract(Mem[Adr*BlockSize+a+T]);
    if (a < (long)Mem[Adr*BlockSize+BlockSize-1])
      Write(Mem[Adr*BlockSize+a]);
    }
}

void ExctractData() {
  InitQueue();
  BTreeExctract(Root);
}


/***********************************************************/
/****                   Main                            ***/
/***********************************************************/

void RUN(long t) {
  N = t;

  printf("BTREE VIR   N = %ld, m = %ld, B = %ld\n",(long)N,(long)m,(long)BlockSize);
  MakeData();

  Mem.Init();

  InitBTree();
  InsertData();
  ExctractData();
  Mem.Flush();
  printf("Nodes used = %ld\n\n",LastNode);

  Mem.WriteStatus();
  WriteStatusFile("btreesvi.out",N);
  printf("\n\n");

}

void main () {
  if (BlockSize % 2 != 0)
    Fejl("Blocksize must be even\n");

  printf("Heap-sorting with BTree, Virtuel memory\n");

        RUN(100);
        RUN(500);
        RUN(1000);
        RUN(5000);
        for(int qq = 10000; qq <= 100000; qq += 10000)
          RUN(qq);

}
```

btree-external.c : External memory version of B-tree heapsort

```
#include <stdio.h>
#include <stdlib.h>
#include "ExtMem.h"

#define MaxSize 100000
#define m NrOfBlocksInMem

#define T (BlockSize-1)/2 /* The degree of the B-tree, size of the B-node */
```

```
#define NIL -1

#define min(a,b) (a < b) ? a : b

ExtMem Mem((long)10*MaxSize);
long N;
long LastNode;
long Root;       // Roots external adress
long ExtAdr[m];  // The internal blocks, external adresses
long Priority[m];
long Check = 0;

TYPE  Data[MaxSize];
int  Found[MaxSize];

void SwapData(long t1, long t2) {
  TYPE t = Data[t1];
  Data[t1] = Data[t2];
  Data[t2] = t;
}

void MakeData() {
  long a;
  TYPE t;
  randomize;
  for(a = 0; a < N; a++)
    Data[a] = a;
  for(a = 0; a < N; a++)
    SwapData(random(N),random(N));
}

/*********************************************************/
/****                      Diverse                    ***/
/*********************************************************/

int GetPos(long Adr) {    // Returns the internal blocknumber
  int a;
  int P = -1;
  for(a = 0; a < m; a++)
    if (ExtAdr[a] == Adr) P = a;
  if (P == -1) {
    // Not found
    for(a = 0; a < m; a++)
      if (Priority[a] == m) P = a;
    Mem.BlockWrite(P,ExtAdr[P]);
    Mem.BlockRead(Adr,P);
    ExtAdr[P] = Adr;
    // Correct priorities
    for(a = 0; a < m; a++)
      if (a != P) {
        if (ExtAdr[a] != Root)
          Priority[a]++;
        }
      else
        Priority[a] = 2;
    }
  else {
    // Found
    if (ExtAdr[P] != Root) {
      for(a = 0; a < m; a++)
        if ((Priority[a] < Priority[P]) && (ExtAdr[a] != Root))
          Priority[a]++;
```

```
      Priority[P] = 2;
      }
    }
  return P;
}

int SizePos(long Adr) {
  return GetPos(Adr)*BlockSize+BlockSize-1;
}

void ClearBNode(long Adr) {
  int a;
  for(a = 0; a < T; a++)
    Mem[GetPos(Adr)*BlockSize+a] = 0;
  for(a = 0; a <=T; a++)
    Mem[GetPos(Adr)*BlockSize+T+a] = NIL;
  Mem[SizePos(Adr)] = 0;
}

void MakeNewBNode(long& Adr) {
  int a;
  for(a = 0; a < m; a++)
    if (Priority[a] == m) {
      if (ExtAdr[a] != NIL)
        Mem.BlockWrite(a,ExtAdr[a]);
      Priority[a] = 2;
      ExtAdr[a] = ++LastNode;
      Adr = LastNode;
      ClearBNode(LastNode);
      }
    else if (Priority[a] != 1)
      Priority[a]++;
}

void MakeNewRoot() {
  int a;
  for(a = 0; a < m; a++)
    if (Priority[a] == m) {
      if (ExtAdr[a] != NIL)
        Mem.BlockWrite(a,ExtAdr[a]);
      Priority[a] = 1;
      ExtAdr[a] = ++LastNode;
      Root = LastNode;
      ClearBNode(Root);
      }
    else
      Priority[a]++;
}

/*********************************************************/
/****                  Init and save                  ***/
/*********************************************************/

void InitBTree() {
  Root = NIL;
  LastNode = -1;
  for(int a = 0; a < m; a++) {
    ExtAdr[a] = NIL;
    Priority[a] = a+1;
    }
}
```

```
void SaveBTree() {
  for(int a = 0; a < m; a++)
    if (ExtAdr[a] != NIL)
      Mem.BlockWrite(a,ExtAdr[a]);
}

/*******************************************************/
/****                    Insert                     ***/
/*******************************************************/

void InsertSorted(long Adr, TYPE X, long NewPtr) {
  int a;

  for(a = (long)Mem[SizePos(Adr)]-1; (a >= 0) && (Mem[GetPos(Adr)*BlockSize+a] > X); a--) {
    Mem[GetPos(Adr)*BlockSize+a+1] = Mem[GetPos(Adr)*BlockSize+a];
    Mem[GetPos(Adr)*BlockSize+a+2+T] = Mem[GetPos(Adr)*BlockSize+a+T+1];
    }
  a++;
  Mem[GetPos(Adr)*BlockSize+a] = X;
  Mem[GetPos(Adr)*BlockSize+a+T+1] = NewPtr;
  ++Mem[SizePos(Adr)];
}

long FindRightChild(long Adr, TYPE X) {
  long P = 0;
  while ((Mem[GetPos(Adr)*BlockSize+P] < X) && (P < (long) Mem[SizePos(Adr)]))
    P++;
  P = Mem[GetPos(Adr)*BlockSize+T+P];
  return P;
}

void SplitNode(long Adr, long& NewNode, TYPE& SplitE) {
  int a,b;
  MakeNewBNode(NewNode);
  a = T/2+1;
  Mem[SizePos(Adr)] = a-1;
  Mem[SizePos(NewNode)] = T-a;
  b = 0;
  while (a < T) {
    Mem[GetPos(NewNode)*BlockSize+b] = Mem[GetPos(Adr)*BlockSize+a];
    Mem[GetPos(Adr)*BlockSize+a] = 0;
    Mem[GetPos(NewNode)*BlockSize+b+T] = Mem[GetPos(Adr)*BlockSize+a+T];
    Mem[GetPos(Adr)*BlockSize+a+T] = NIL;
    a++; b++;
    }
  Mem[GetPos(NewNode)*BlockSize+b+T] = Mem[GetPos(Adr)*BlockSize+a+T];
  Mem[GetPos(Adr)*BlockSize+a+T] = NIL;

  SplitE = Mem[GetPos(Adr)*BlockSize+T/2];
  Mem[GetPos(Adr)*BlockSize+T/2] = 0;
}

void SplitRoot(long& Adr, TYPE X) {
  long NewNode;
  TYPE SplitE;

  SplitNode(Adr,NewNode,SplitE);
  // Make new root
  MakeNewRoot();
  Mem[GetPos(Root)*BlockSize] = SplitE;
  Mem[GetPos(Root)*BlockSize+T] = Adr;
  Mem[GetPos(Root)*BlockSize+T+1] = NewNode;
```

```
  Mem[SizePos(Root)] = 1;
  if (X > SplitE)
    Adr = NewNode;
}

void SplitInternalNode(long& Adr, long Father, TYPE X) {
  long NewNode;
  TYPE SplitE;

  SplitNode(Adr,NewNode,SplitE);
  InsertSorted(Father,SplitE,NewNode);
  if (X > SplitE)
    Adr = NewNode;
}

void FindNode(long Adr, TYPE X) {
  long Child;

  if ((long)Mem[SizePos(Adr)] == 0)
    InsertSorted(Adr,X,NIL);
  else {
    if ((Adr == Root) && (Mem[SizePos(Adr)] == T))
      SplitRoot(Adr,X);
    Child = FindRightChild(Adr,X);
    if (Child == NIL)
      InsertSorted(Adr,X,NIL);
    else {
      if (Mem[SizePos(Child)] == T)
        SplitInternalNode(Child,Adr,X);
      FindNode(Child,X);
      }
    }
}

void BtreeInsert(TYPE X) {
  if (Root == NIL)
    MakeNewRoot();
  FindNode(Root,X);
}

/***********************************************************/
/****               Error check                     *****/
/***********************************************************/

void Fejl(char* s) {
  printf(s);
  exit(42);
}

void InsertData() {
  long a,b;
  for(a=0; a < N; a++)
    BtreeInsert(Data[a]);
}

/*************************************************************/
/****                 ExctractData                      ***/
/*************************************************************/

void InitQueue() {
  for(int a = 0; a < m; a++) {
    ExtAdr[a] = NIL;
```

```
    Priority[a] = -1;
    }
  ExtAdr[0] = LastNode+1;
  Priority[0] = 2*N;
  Check = 0;
}

void GetAdr(long Adr,long Depth, long& IntPos) {
  int a,P;
  // Is the depth already there
  P = -1;
  for(a = 1; a < m; a++)
    if (Priority[a] == Depth)
      P = a;
  if (P != -1) {
    // Depth there
    IntPos = P;
    if (ExtAdr[IntPos] != Adr)
      Mem.BlockRead(Adr,IntPos);
    ExtAdr[IntPos] = Adr;  // Only used in debuggin
    }
  else {
    // Depth not there, find a place for it
    P = 0;
    for(a = 1; a < m; a++)
      if (Priority[a] < Priority[P])
        P = a;
    IntPos = P;
    Priority[IntPos] = Depth;
    Mem.BlockRead(Adr,IntPos);
    ExtAdr[IntPos] = Adr;  // Only used in debuggin
    }
}

void Write(TYPE t) {
  Mem[Check % BlockSize] = t;
  //printf("%ld   ",(long)t);
  if (Check % BlockSize == BlockSize-1) {
    Mem.BlockWrite(0,ExtAdr[0]);
    ExtAdr[0]++;
    }
  if (t != Check)
    Fejl("Wrong number\n");
  Check++;
}

void BTreeExctract(long Adr, long Depth) {
  long IntPos;

  for(int a = 0; a <= T; a++) {
    GetAdr(Adr,Depth,IntPos);
    if ((long)Mem[IntPos*BlockSize+a+T] != NIL)
      BTreeExctract(Mem[IntPos*BlockSize+a+T],Depth+1);
    GetAdr(Adr,Depth,IntPos);
    if (a < (long)Mem[IntPos*BlockSize+BlockSize-1])
      Write(Mem[IntPos*BlockSize+a]);
    }
}

void ExctractData() {
  SaveBTree();
  InitQueue();
```

```
  BTreeExctract(Root,0);
}


/**********************************************************/
/****                  Main                           ***/
/**********************************************************/

void RUN(long t) {
  N = t;
  printf("BTREE  N = %ld, m = %ld, B = %ld\n",(long)N,(long)m,(long)BlockSize);
  MakeData();

  Mem.Init();

  InitBTree();
  InsertData();
  ExctractData();
  printf("Nodes used = %ld\n\n",LastNode);

  Mem.WriteStatus();
  WriteStatusFile("btreesor.out",N);
  printf("\n\n");
}

void main () {

  if (BlockSize % 2 != 0)
    Fejl("Blocksize must be even\n");

  printf("Heap-sorting with BTree\n");

        RUN(100);
        RUN(500);
        RUN(1000);
        RUN(5000);
        for(int qq = 10000; qq <= 100000; qq += 10000)
          RUN(qq);

}
```

heapsort.c : Binary heapsort and I/O-queue heapsort

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "ExtMem.h"
#include "VirMem.h"

#define InputSize 100000

/* 'I/O-optimal' priority queue

         Two buffers: One for insertions and one for delete-max's

         When sorting only one buffer is needed at the time,
         therefore there is no actual root while inserting, the root
         is generated before the first delete-max

         Each internal node contains the indices of the children nodes.

         The algortihm requires certain proportions (h is an integer):
```

```
         N, M & fanout = h * B
         fanout (CPN) : (M-B)/(B+1) and is multiplum of B
                 (M records, B are the out-buffer when merging. The remaing records,
                 M-B, are divided among the children. Each child occupies B+1 records
                 B as in-buffer when merging and 1 the keep the index. Therefore
                 (M-B)/(B+1) children.
         M=CPN(B+1)+B
         m (NrOfBlocksInMem) = M/B
         B=(M-h)/(h+1)
*/

ExtMem Mem(2*InputSize+3*NrOfBlocksInMem*BlockSize);
VirMem VMem(InputSize);
TYPE Input[InputSize];
int Leaf[2*(InputSize/BlockSize)];
signed long HeapEnd;
signed long root_inds[NrOfBlocksInMem];


signed long B=BlockSize;
signed long N=InputSize;
signed long m=NrOfBlocksInMem;
signed long M=m*B;
signed long CPN=(M-B)/(B+1);   /* Children Per Node */
signed long RPN=((M-CPN)/B)*B;   /* Records Per Node */
signed long rpn=RPN/B;  /* number of Records blocks Per Node */

signed long Extr_End=2*N;
signed long Intern_i, Extern_i, Extern_m;

void make_input() {
        long i, i1, i2;
        TYPE temp;
        time_t t;

        for (i=0; i<N; i++) Input[i]=(TYPE) i;
        srand((unsigned) time(&t));
        for (i=0; i<N; i++){
                i1=rand()%(N-1); i2=rand()%(N-1);
                temp=Input[i1]; Input[i1]=Input[i2]; Input[i2]=temp;
        }
}

void init() {
}

void done() {
}

void swap(long i, long j) {
        TYPE Temp;
        Temp = Mem[i]; Mem[i] = Mem[j]; Mem[j] = Temp;
}

void QuickSort(long begin, long end) {
        long i,j;
        TYPE Piv,Temp;

        if (end - begin == 1) {
                if (Mem[begin]>Mem[end]) swap(begin, end);
        }
        else if (end - begin > 0) {
```

```
                Piv = Mem[begin];
                j = end;
                i = begin+1;
                while (i < j) {
                        if (Mem[i] > Piv) {
        while ((Mem[j] > Piv) && (i < j))
                j--;
        if (i < j) swap(i,j);
                        }
                        else i++;
                }
                if (Mem[i] <= Piv)
                        swap(begin,i);
                else {
                        i--; swap(begin,i);
                }
                QuickSort(begin,i-1);
                QuickSort(i+1,end);
        }
}

void InternalSort() {
        QuickSort(0, Intern_i);
}

void Shift_Up(signed long p, signed long c, signed long csize) {
  signed long ip=0, ic=0, i=0;
  signed long end_p=RPN-1, end_m;
  int stop;

  while (p>-1) {
    if (csize==-1 || end_p==-1) {
      printf("I/O-Shift_up: indekser =-1");
      exit(1);
    }
    ic=ip=0;
    i=0;
    Mem.BlockRead(p*m, 0); /* parents first block */
    Mem.BlockRead(c*m,1);  /* the childs first block */
    /* Two-way merge using 3 blocks (0, 1 and 2) */
    stop=0;
    while (!stop) {
      if ( (ip<=end_p) && ( (ic>csize) || (Mem[ip%B]<Mem[B+ic%B]) ) ) {
        Mem[i%B+2*B]=Mem[ip%B]; ip++;
        if ((ip<=end_p) && (ip%B==0))
          Mem.BlockRead(p*m+ip/B, 0);
        } else {
          Mem[i%B+2*B]=Mem[ic%B+B]; ic++;
          if ((ic<=csize) && (ic%B==0))
            Mem.BlockRead(c*m+ic/B, 1);
        }
        i++;
        if (i%B==0)
          Mem.BlockWrite(2, (Extr_End+i)/B-1);
        else if (i==csize+1) {
          i=(csize/B+1)*B;
          Mem.BlockWrite(2, (Extr_End+i)/B-1);
        } else if (ip>end_p && ic>csize)
          stop=1;
        }
          if (i%B!=0)
            Mem.BlockWrite(2, (Extr_End+i)/B);
```

```
          csize=(csize/B+1)*B-1;
          end_p=(end_p/B+1)*B-1;
          for(i=0; i<=csize/B; i++) { /* Whole blocks of the merged to the child node */
            Mem.BlockRead(Extr_End/B+i, 0); Mem.BlockWrite(0, c*m+i);
          }
          for (i=0; i<=end_p/B; i++) {
            Mem.BlockRead((csize+1+Extr_End)/B+i, 0); Mem.BlockWrite(0, p*m+i);
          }
        c=p; csize=RPN-1;
        p=c/CPN-1; end_p=RPN-1;
        }
}

void Flush() {
        signed long i;
        signed long p, c, IndexBlock, BlockOfSet;

        InternalSort();
        for (i=0; i<m; i++)
                Mem.BlockWrite(i, ++Extern_i);
        Extern_m++;
        /* If parent node is not the 'not existing root' then
                 assign the children index to parent and shift-up */
        p=Extern_m/CPN-1;
        c=Extern_m%CPN;
        if (p>-1) {
          Leaf[p]=0;                  /* Parent is no longer a leaf */
          IndexBlock=(RPN+c)/B;
          Mem.BlockRead(p*m+IndexBlock, 0);  /* Get parents indices */

                BlockOfSet=(RPN+c)%B;
                Mem[BlockOfSet]=Intern_i;
                Mem.BlockWrite(0, p*m+IndexBlock);

                Shift_Up(p, Extern_m, Intern_i);
        } else
                root_inds[c]=Intern_i;

        Intern_i=-1;
}

void Insert(TYPE x) {
        if (Intern_i==RPN-1)
                Flush();
        Mem[++Intern_i]=x;
}

void Refill(signed long p, TYPE *psize) {
        int stop=0, lonely=1;
        signed long i, c, max, ins=CPN-1;
        signed long inds[NrOfBlocksInMem];

        if (*psize>-1) {
                for (c=0; c<=*psize/B; c++)  /* Save eventual records in node */
                        Mem.BlockRead(p*m+c, c);
                for (c=0; c<=*psize; c++)
                        Mem[RPN-1-c]=Mem[*psize-c];
                for (c=0; c<=*psize/B; c++)
                        Mem.BlockWrite(rpn-1-c, Extr_End/B+m+c);
                for (c=0; c<B; c++)
                        Mem[RPN-1-c]=Mem[RPN-1-(*psize/B)*B-c];
        }
```

```
	if (p>-1) for (c=rpn; c<m; c++)  /* Get indices */
		Mem.BlockRead(p*m+c, c);
	for (c=0; c<CPN; c++) {
		inds[c]=Mem[RPN+c];  /* Move indices to local variable */
		if (inds[c]>-1)      /* and load first blocks of input */
			Mem.BlockRead((p+1)*CPN*m+c*m+inds[c]/B, c);
		else ins-=1;
	}
	i=*psize;
	while (!stop) {
		i++; max=0;
		while (inds[max]==-1) max++;
		for (c=max; c<CPN; c++)  /* find index to max element */
			if (inds[c]>-1 && Mem[c*B+inds[c]%B]>Mem[max*B+inds[max]%B])
				max=c;
		Mem[RPN-i%B-1]=Mem[max*B+inds[max]%B]; /* move max element to output block */
		if ((i+1)%B==0)
			if (p==-1) Mem.BlockWrite(rpn-1, (Extr_End+i+1)/B-1);
			else Mem.BlockWrite(rpn-1, (Extr_End+i+1)/B+m-1);
		/* if input node empty or parent is full then stop. */
		if (inds[max]==0) ins-=1;
		if (i==RPN-1 || ins==-1 ||
				(inds[max]==0 && !Leaf[CPN*(p+1)+max])) stop=1;
		else if (inds[max]>0 && inds[max]%B==0)
			Mem.BlockRead((p+1)*CPN*m+max*m+(inds[max]-1)/B, max);
		--inds[max];
	};
	*psize=(TYPE) i;
	if ((i+1)%B!=0) {  /* adjust i to nearest whole multiplum of B etc */
		i=(i/B+1)*B-1;
		if (p==-1) Mem.BlockWrite(rpn-1, (Extr_End+i+1)/B-1);
		else Mem.BlockWrite(rpn-1, (Extr_End+i+1)/B+m-1);
	}
	if (p>-1) {
		for (c=0; c<=i/B; c++)  /* Move records to node */
			Mem.BlockRead(Extr_End/B+c+m, (i+1)/B-c-1);
		if ((i-*psize)>0)
			for (c=0; c<=*psize; c++)
				Mem[c]=Mem[c+(i-*psize)];
		for (c=0; c<=i/B; c++)  /* Move records to node */
			Mem.BlockWrite(c, p*m+c);
	}
	for (c=0; c<CPN; c++)    /* Refill underfull children */
		if (inds[c]<(RPN-1)/2 && !Leaf[CPN*(p+1)+c])
			Refill(CPN*(p+1)+c, (TYPE *) &inds[c]);
	if (p==-1) {
		for (c=0; c<=i/B; c++)    /* Load root records */
			Mem.BlockRead(Extr_End/B+c, (i+1)/B-c-1);
		if ((i-*psize)>0)
			for (c=0; c<=*psize; c++)
				Mem[c]=Mem[c+(i-*psize)];
	}
	for (c=0; c<CPN; c++) {  /*     Load indices */
		Mem[RPN+c]=inds[c];
		lonely&=(inds[c]==-1);
	}
	if (p>-1 && lonely)
		Leaf[p]=1;
	if (p>-1) for (c=rpn; c<m; c++) /* Move indices to node */
			Mem.BlockWrite(c, p*m+c);
}
```

```
TYPE Delete_Min() {
        if (Intern_i==-1)
                Refill(-1, (TYPE *) &Intern_i);
        return Mem[Intern_i--];
}

void build_heap() {
        signed long i;

        for (i=0; i<2*InputSize/BlockSize; i++) Leaf[i]=1;

        Intern_i=Extern_i=Extern_m=-1;
        for (i=RPN; i<M; i++) Mem[i]=-1;
        for (i=0; i<NrOfBlocksInMem; i++) root_inds[i]=-1;

        for (i=0; i<N; i++)
                Insert(Input[i]);
        Flush();
        for(i=RPN; i<M; i++)
                Mem[i]=root_inds[i-RPN];
}

void IO_opt_heapsort(signed long n) {
        long i, j;

        N=n;

        make_input();
  Mem.Init();

        build_heap();

        for (i=0; i<N; i++) Input[i]=0;

        for (i=0; i<N; i++)
                Input[i]=(TYPE)Delete_Min();

        Mem.WriteStatus(); printf("\n");
        WriteStatusFile("extheap.out",N);

        for (i=0; i<N; i++)
                if (Input[i]!=N-1-i) {
                        printf("I/O-optimal ikke sorteret. Pos = %ld\n", (signed long) i);
      exit(1);
                }
}

/******************* d-ary heapsort *****************/

unsigned long maxchild (unsigned long x)
{ unsigned long c, c2;

        c2=(c=2*x+1)+1;
        if (c>HeapEnd) return 0;
        else if (c2<=HeapEnd && VMem[c2]>VMem[c]) return c2;
                         else return c;
}

void shift_down(unsigned long x)
{ unsigned long child=maxchild(x);
        TYPE temp;
```

```
        if (child>0) {
                temp=VMem[x];
                if (VMem[child]>temp) {
                        while ( child>0 && VMem[child]>temp ) {
        VMem[x]=VMem[child];
        x=child;
        child=maxchild(x);
                        }
                        VMem[x]=temp;
                }
        }
}

void remove_max()
{ TYPE temp=VMem[HeapEnd];
        long x=0, child;

        VMem[HeapEnd--]=VMem[0];
        child=maxchild(x);
        while ( (HeapEnd>0) && (child>0) && (VMem[child]>temp) ) {
                VMem[x]=VMem[child];
                x=child;
                child=maxchild(x);
        }
        VMem[x]=temp;
}

void heapsort(signed long n)
{ long i;

        N=n;

        make_input();
        VMem.Init();

        for (i=0; i<N; i++)
                VMem[i]=Input[i];

        HeapEnd=N-1;
        for (i=(long)HeapEnd/2; i>-1; i--)
                shift_down(i);
        while(HeapEnd>0) /*for (i=0; i<N; i++)*/
                remove_max();

        VMem.Flush();
        VMem.WriteStatus(); printf("\n");
        WriteStatusFile("heapsort.out",N);

        for (i=1; i<N; i++)
                if (!(VMem[i]==i))
                        printf("heapsort ikke sorteret\n");
}


void RUN(long t) {
        printf("HEAPSORT :  N=%ld  B=%ld  M=%ld  CPN=%ld  RPN=%ld\n",
                                 (long) t, (long) B, (long) M, (long)CPN, (long)RPN);
        IO_opt_heapsort(t);
        heapsort(t);
}
```

```
void main () {

        init();

        RUN(100);
        RUN(500);
        RUN(1000);
        RUN(5000);
        for(int qq = 10000; qq <= 100000; qq += 10000)
          RUN(qq);

        done();
}
```

Appendix D

The priority-queue programs

queues.c B-heap and B-tree in virtual memory plus I/O-queue in external memory.

queues.c : Three priority queues

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "VirMem.h"
#include "ExtMem.h"

#define MaxSize 100000

#define T (BlockSize-1)/2   /* The degree of the B-tree, size of the B-node */
#define NIL -1

#define min(a,b) (a < b) ? a : b

long N;
int alpha;  /* In percent */

VirMem BTreeMem(10*MaxSize);
ExtMem IOMem(4*MaxSize+4*NrOfBlocksInMem*BlockSize);
VirMem HeapMem(2*MaxSize);

int leaf[2*(MaxSize/BlockSize)];
signed long sizes[2*(MaxSize/BlockSize)];
signed long HeapEnd;
signed long root_inds[NrOfBlocksInMem];

TYPE StartData[MaxSize];
TYPE OpData[MaxSize];
int Operations[MaxSize];



signed long B=BlockSize;
signed long m=NrOfBlocksInMem;
signed long M=m*B;
signed long CPN=m/2-1;   /* Children Per Node */
signed long RPN=CPN*B;   /* Records Per Node */
signed long rpn=RPN/B;  /* number of Records blocks Per Node */
signed long nodesize=(signed long)rpn+(signed long)ceil((double)CPN/B);

signed long Extr_End=4*MaxSize;
signed long ins=CPN;
```

```
signed long INS=CPN*B;
signed long inds=2*CPN;
signed long INDS=inds*B;
signed long outbuf=m-1;
signed long OUTBUF=outbuf*B;
signed long levels, newleaf, next_to_melt;
signed long maxi, ini, recs;
int ins_saved;




void MakeStartData() {
  long a;
  for(a = 0; a < MaxSize; a++) {
    StartData[a] = random(MaxSize);
    OpData[a] = random(MaxSize);
    Operations[a] = random(100);
    }
}


/*******************************************************/
/*****            BTrees                         *****/
/*******************************************************/

long BTreeLastNode;
long BTreeRoot;         // Roots blocknumber
long BTreeBufferCount;  // The internal buffercount
long BTreeBufferSize;   // The last element in the buffer

void InitBTree() {
  BTreeRoot = NIL;
  BTreeLastNode = 0;  // Gives room so the buffer can be in block 0
  BTreeBufferCount = 0;
  BTreeBufferSize = 0;
  BTreeMem.Init();
}

int BTreeSizePos(long Adr) {
  return Adr*BlockSize+BlockSize-1;
}

void BTreeTouchBuffer() {
  BTreeMem[0] = BTreeMem[0];
}

/*****                  Insert                       *****/

void ClearBNode(long Adr) {
  int a;
  for(a = 0; a < T; a++)
    BTreeMem[Adr*BlockSize+a] = 0;
  for(a = 0; a <=T; a++)
    BTreeMem[Adr*BlockSize+T+a] = NIL;
  BTreeMem[BTreeSizePos(Adr)] = 0;
}

void MakeNewBNode(long& Adr) {
  Adr = ++BTreeLastNode;
  ClearBNode(Adr);
```

```
}

void BTreeMakeNewRoot() {
  BTreeRoot = ++BTreeLastNode;
  ClearBNode(BTreeRoot);
}

void BTreeInsertSorted(long Adr, TYPE X, long NewPtr) {
  int a;

  for(a = (long)BTreeMem[BTreeSizePos(Adr)]-1; (a >= 0) && (BTreeMem[Adr*BlockSize+a] > X); a--) {
    BTreeMem[Adr*BlockSize+a+1] = BTreeMem[Adr*BlockSize+a];
    BTreeMem[Adr*BlockSize+a+2+T] = BTreeMem[Adr*BlockSize+a+T+1];
    }
  a++;
  BTreeMem[Adr*BlockSize+a] = X;
  BTreeMem[Adr*BlockSize+a+T+1] = NewPtr;
  ++BTreeMem[BTreeSizePos(Adr)];
}

long BTreeFindRightChild(long Adr, TYPE X) {
  long P = 0;
  while ((BTreeMem[Adr*BlockSize+P] < X) && (P < (long) BTreeMem[BTreeSizePos(Adr)]))
    P++;
  P = BTreeMem[Adr*BlockSize+T+P];
  return P;
}

void BTreeSplitNode(long Adr, long& NewNode, TYPE& SplitE) {
  int a,b;
  MakeNewBNode(NewNode);
  a = T/2+1;
  BTreeMem[BTreeSizePos(Adr)] = a-1;
  BTreeMem[BTreeSizePos(NewNode)] = T-a;
  b = 0;
  while (a < T) {
    BTreeMem[NewNode*BlockSize+b] = BTreeMem[Adr*BlockSize+a];
    BTreeMem[Adr*BlockSize+a] = 0;
    BTreeMem[NewNode*BlockSize+b+T] = BTreeMem[Adr*BlockSize+a+T];
    BTreeMem[Adr*BlockSize+a+T] = NIL;
    a++; b++;
    }
  BTreeMem[NewNode*BlockSize+b+T] = BTreeMem[Adr*BlockSize+a+T];
  BTreeMem[Adr*BlockSize+a+T] = NIL;

  SplitE = BTreeMem[Adr*BlockSize+T/2];
  BTreeMem[Adr*BlockSize+T/2] = 0;
}

void BTreeSplitRoot(long& Adr, TYPE X) {
  long NewNode;
  TYPE SplitE;

  BTreeSplitNode(Adr,NewNode,SplitE);
  // Make new root
  BTreeMakeNewRoot();
  BTreeMem[BTreeRoot*BlockSize] = SplitE;
  BTreeMem[BTreeRoot*BlockSize+T] = Adr;
  BTreeMem[BTreeRoot*BlockSize+T+1] = NewNode;
  BTreeMem[BTreeSizePos(BTreeRoot)] = 1;
  if (X > SplitE)
    Adr = NewNode;
```

```
}

void BTreeSplitInternalNode(long& Adr, long Father, TYPE X) {
  long NewNode;
  TYPE SplitE;

  BTreeSplitNode(Adr,NewNode,SplitE);
  BTreeInsertSorted(Father,SplitE,NewNode);
  if (X > SplitE)
    Adr = NewNode;
}

void BTreeFindNode(long Adr, TYPE X) {
  long Child;

  BTreeTouchBuffer();
  if ((long)BTreeMem[BTreeSizePos(Adr)] == 0)
    BTreeInsertSorted(Adr,X,NIL);
  else {
    if ((Adr == BTreeRoot) && (BTreeMem[BTreeSizePos(Adr)] == T))
      BTreeSplitRoot(Adr,X);
    Child = BTreeFindRightChild(Adr,X);
    if (Child == NIL)
      BTreeInsertSorted(Adr,X,NIL);
    else {
      if (BTreeMem[BTreeSizePos(Child)] == T)
        BTreeSplitInternalNode(Child,Adr,X);
      BTreeFindNode(Child,X);
      }
    }
}

int BTreeInBuffer(TYPE X) {
  if ((BTreeBufferCount == BTreeBufferSize) ||
      ((long) X <=  (long) BTreeMem[BTreeBufferSize-1]))
    return 0;
  else
    return 1;
}

void BTreeInsertInBuffer(TYPE X) {
  TYPE temp[BlockSize];  // Not nescecarry, but makes programming easier
  long a,b,c;
  // Insert element
  c = BTreeBufferSize-BTreeBufferCount+1;
  a = BTreeBufferCount;
  for(b = 0; b < c; b++) {
    if ((BTreeMem[a] > X) && (a < BTreeBufferSize))
      temp[b] = BTreeMem[a++];
    else {
      temp[b] = X;
      X = -1000;
      }
    }
  for(a = 0; a < c; a++)
    BTreeMem[a] = temp[a];
  BTreeBufferCount = 0;
  BTreeBufferSize = c;
  // Check if buffer full
  if (BTreeBufferSize == BlockSize) {
    for(; BTreeBufferSize > BlockSize/2;) {
      BTreeBufferSize--;
```

```
      BTreeFindNode(BTreeRoot,BTreeMem[BTreeBufferSize]);
      }
    }
}

void BTreeInsert(TYPE X) {
  if (BTreeInBuffer(X))
    BTreeInsertInBuffer(X);
  else {
    if (BTreeRoot == NIL)
      BTreeMakeNewRoot();
    BTreeFindNode(BTreeRoot,X);
    }
}


/***********      Delete max BTree                 ******/

void BTreeCopyBufferRoot() {
  // Copy elements
  BTreeBufferCount = 0;
  long Size = BTreeMem[BTreeSizePos(BTreeRoot)];
  for(long a = Size-1; a >= 0; a--) {
    BTreeMem[BTreeBufferCount++] = BTreeMem[BTreeRoot*BlockSize+a];
    BTreeBufferSize++;
    }
  BTreeRoot = NIL;
}

void BTreeCopyBuffer(long Adr, long Child) {
  // Copy child elements
  BTreeBufferCount = 0;
  long Size = BTreeMem[BTreeSizePos(Child)];
  for(long a = Size-1; a >= 0; a--) {
    BTreeMem[BTreeBufferCount++] = BTreeMem[Child*BlockSize+a];
    BTreeBufferSize++;
    }
  // Copy element from father node
  long Last = BTreeMem[BTreeSizePos(Adr)];
  BTreeMem[BTreeBufferCount] = BTreeMem[Adr*BlockSize+Last-1];
  --BTreeMem[BTreeSizePos(Adr)];
  BTreeBufferSize++;
  Child = NIL;
  // This part is not nescesarry, but makes debugging easier
  BTreeMem[Adr*BlockSize+Last-1] = 0;
  BTreeMem[Adr*BlockSize+Last+T] = NIL;
}

void BTreeSteal(long Adr,long Child,long Child2) {
  // Make room in Child
  long a;
  for(a = T-2; a >= 0; a--) {
    BTreeMem[Child*BlockSize+a+1] = BTreeMem[Child*BlockSize+a];
    BTreeMem[Child*BlockSize+a+2+T] = BTreeMem[Child*BlockSize+a+T+1];
    }
  BTreeMem[Child*BlockSize+a+2+T] = BTreeMem[Child*BlockSize+a+T+1];
  // Copy element from Adr
  long SizeAdr = BTreeMem[BTreeSizePos(Adr)];
  BTreeMem[Child*BlockSize] = BTreeMem[Adr*BlockSize+SizeAdr-1];
  ++BTreeMem[BTreeSizePos(Child)];
  // Insert elemet from child2 in Adr
  long SizeChild2 = BTreeMem[BTreeSizePos(Child2)];
```

```
  BTreeMem[Adr*BlockSize+SizeAdr-1] = BTreeMem[Child2*BlockSize+SizeChild2-1];
  BTreeMem[Child2*BlockSize+SizeChild2-1] = 0;
  --BTreeMem[BTreeSizePos(Child2)];
  // Move pointer from Child2 to Child
  BTreeMem[Child*BlockSize+T] = BTreeMem[Child2*BlockSize+SizeChild2+T];
  BTreeMem[Child2*BlockSize+SizeChild2+T] = NIL;
}

void BTreeMelt(long Adr,long Child,long Child2) {
  long Count = BTreeMem[BTreeSizePos(Child2)];
  long Size = BTreeMem[BTreeSizePos(Child)];
  long SizeAdr = BTreeMem[BTreeSizePos(Adr)];
  // Copy from Adr
  BTreeMem[Child2*BlockSize+Count] = BTreeMem[Adr*BlockSize+SizeAdr-1];
  BTreeMem[Adr*BlockSize+SizeAdr-1] = 0;
  BTreeMem[Adr*BlockSize+SizeAdr+T] = NIL;
  --BTreeMem[BTreeSizePos(Adr)];
  ++BTreeMem[BTreeSizePos(Child2)];
  Count++;
  // Copy from Child
  long a;
  for(a = 0; a < Size; a++) {
    BTreeMem[Child2*BlockSize+Count] = BTreeMem[Child*BlockSize+a];
    BTreeMem[Child2*BlockSize+Count+T] = BTreeMem[Child*BlockSize+a+T];
    ++BTreeMem[BTreeSizePos(Child2)];
    Count++;
    }
  BTreeMem[Child2*BlockSize+Count+T] = BTreeMem[Child*BlockSize+a+T];
  // Check root
  if ((Adr == BTreeRoot) && (BTreeMem[BTreeSizePos(Adr)] == 0))
    BTreeRoot = Child2;
}

void BTreeCorrectSize(long Adr, long Child, long Child2) {
  long Size2 = BTreeMem[BTreeSizePos(Child2)];
  if (Size2 > T/2)
    BTreeSteal(Adr,Child,Child2);
  else
    BTreeMelt(Adr,Child,Child2);
}

void BTreeGetNewBuffer(long Adr) {
  BTreeTouchBuffer();
  long Size = BTreeMem[BTreeSizePos(Adr)];
  long Child = BTreeMem[Adr*BlockSize+Size+T];
  if (Child == NIL)
    BTreeCopyBufferRoot();
  else if (BTreeMem[Child*BlockSize+T] == NIL) {
    BTreeCopyBuffer(Adr,Child);
    // Check root
    if ((Adr == BTreeRoot) && (BTreeMem[BTreeSizePos(Adr)] == 0)) {
      BTreeRoot = BTreeMem[Adr*BlockSize+T];
      BTreeMem[Adr*BlockSize+T] = NIL;
      }
    }
  else {
    BTreeGetNewBuffer(Child);
    BTreeTouchBuffer();
    Size = BTreeMem[BTreeSizePos(Adr)];
    Child = BTreeMem[Adr*BlockSize+Size+T];
    long Child2 = BTreeMem[Adr*BlockSize+Size+T-1];
    if ((Child != NIL) && (BTreeMem[BTreeSizePos(Child)] < T/2))
```

```
      BTreeCorrectSize(Adr,Child,Child2);
    }
}

void BTreeDeleteMax() {
  if (BTreeBufferCount == BTreeBufferSize) {
    BTreeBufferSize = 0;
    BTreeGetNewBuffer(BTreeRoot);
    BTreeBufferCount = 0;
    }
  BTreeBufferCount++;
}


/************************************************************/
/*****              B-heaps                               ***/
/************************************************************/


unsigned long HeapMaxchild (unsigned long x)
{ unsigned long c, c2, i, max;

        c=B*x+1;
        c2=B*(x+1);
        if (c2>HeapEnd) c2=HeapEnd;
        if (c>HeapEnd) return 0;
        max=c;
        for (i=c+1; i<=c2; i++)
                if (HeapMem[max]<HeapMem[i])
                        max=i;
        return max;
}

void HeapShift_down(unsigned long x)
{ unsigned long child=HeapMaxchild(x);
        TYPE temp;

        if (child>0) {
                temp=HeapMem[x];
                if (HeapMem[child]>temp) {
                        while ( child>0 && HeapMem[child]>temp ) {
                                HeapMem[x]=HeapMem[child];
                                x=child;
                                child=HeapMaxchild(x);
                        }
                        HeapMem[x]=temp;
                }
        }
}

TYPE HeapDelete_max()
{ TYPE ret, temp;
        long x=0, child;

        if (HeapEnd<0) {
                printf("Delete-max p tom B-hob");
                exit(1);
        }
        temp=HeapMem[HeapEnd--];
        ret=HeapMem[0];
        child=HeapMaxchild(x);
        while ( (HeapEnd>0) && (child>0) && (HeapMem[child]>temp) ) {
```

```
                HeapMem[x]=HeapMem[child];
                x=child;
                child=HeapMaxchild(x);
        }
        HeapMem[x]=temp;
        return ret;
}

void HeapShift_up(signed long x)
{ signed long p=(x+B-1)/B-1;
        TYPE temp;
        while ( (x>0) && (HeapMem[x]>HeapMem[p]) ) {
                temp=HeapMem[x]; HeapMem[x]=HeapMem[p]; HeapMem[p]=temp;
                x=p;
                p=(x+B-1)/B-1;
        }
}

void HeapInsert(TYPE k)
{
        HeapMem[++HeapEnd]=k;
        HeapShift_up(HeapEnd);
}

void HeapInit()
{
        HeapMem.Init();
        HeapEnd=-1;
}






/****************************************************************/
/*****             I/O-queue                                 ***/
/****************************************************************/




void IOswap(long i, long j) {
        TYPE Temp;
        Temp = IOMem[i]; IOMem[i] = IOMem[j]; IOMem[j] = Temp;
}

void IOQuickSort(long begin, long end) {
        long i,j;
        TYPE Piv,Temp;

        if (end - begin == 1) {
                if (IOMem[begin]>IOMem[end]) IOswap(begin, end);
        }
        else if (end - begin > 0) {
                Piv = IOMem[begin];
                j = end; i = begin+1;
                while (i < j) {
                        if (IOMem[i] > Piv) {
                                while ((IOMem[j] > Piv) && (i < j))
```

```
                                        j--;
                                if (i < j) IOswap(i,j);
                        }
                        else i++;
                }
                if (IOMem[i] <= Piv) IOswap(begin,i);
                else  {             i--; IOswap(begin,i); }
                IOQuickSort(begin,i-1);
                IOQuickSort(i+1,end);
        }
}

signed long IOGet_size(signed long x)
{ signed long p, c, IndexBlock;
        if (x>2*N/B) {
                printf("Get_size: x>2*n"); exit(1);
        }
        if (x==-1) return maxi;
        BlockReads++;
        return sizes[x];
}

void IOSet_size(signed long x, TYPE s)
{ signed long p, c, IndexBlock;
        if (x>2*N/B) {
                printf("Set_size: x>2*n"); exit(1);
        }
        if (x==-1)
                maxi=(signed long) s;
        else {
                sizes[x]=s;
                BlockWrites++;
        }
}

int IOGet_leaf(signed long x)
{
        if (x>2*N/B) {
                printf("Get-leaf: x>2*n"); exit(1);
        }
        if (x<0) {
                printf("Get-leaf: x<0"); exit(1);
        }
        return leaf[x];
}

void IOSet_leaf(signed long x, int s)
{ signed long p, c, IndexBlock;
        if (x>2*N/B) {
                printf("Set_leaf: x>2*n"); exit(1);
        }
        if (x<0) {
                printf("Set-leaf: x<0"); exit(1);
        }
        leaf[x]=s;
}

void IOShift_Up_Insert(signed long c, signed long nsize) {
        signed long ip, ic, i, j, p;
        signed long psize, oldp;
        int stop, leave;
```

```
    p=c/CPN-1;
    while (p>-2) {
      psize=IOGet_size(p);
      if (nsize==-1 || psize==-1) {
        printf("I/O-Shift_up_insert: indekser =-1"); exit(1);
      }
      if (p==-1) for (i=0; i<B; i++) IOMem[INS+i]=IOMem[i];
      else IOMem.BlockRead(p*nodesize, ins); /* parents first block */
      IOMem.BlockRead(c*nodesize+nsize/B, ins+1);  /* the childs last block */
      stop=0; leave=0;
      if (IOMem[(ins+1)*B+nsize%B]<IOMem[INS])  /* If childs largest record */
        leave=stop=1;                           /* than all parent records then leave */
      else IOMem.BlockRead(c*nodesize, ins+1);  /* the childs first block */
      /* Two-way merge using 3 blocks */
      i=ic=ip=0;
      while (!stop) {
        if ( (ip<=psize) && ( (ic>nsize) || (IOMem[INS+ip%B]<IOMem[INS+B+ic%B]) ) ) {
          IOMem[i%B+OUTBUF]=IOMem[INS+ip%B]; ip++;
          if ((ip<=psize) && (ip%B==0)) {
            if (p==-1) for (j=0; j<B; j++) IOMem[INS+j]=IOMem[ip+j];
            else IOMem.BlockRead(p*nodesize+ip/B, ins);
          }
        } else {
          IOMem[i%B+OUTBUF]=IOMem[INS+B+ic%B]; ic++;
          if ((ic<=nsize) && (ic%B==0))
            IOMem.BlockRead(c*nodesize+ic/B, ins+1);
          }
          i++;
          if (i%B==0)
          IOMem.BlockWrite(outbuf, (Extr_End+i)/B+2*m-1);
          else if (i==nsize+1) {
            i=(nsize/B+1)*B;
            IOMem.BlockWrite(outbuf, (Extr_End+i)/B+2*m-1);
          }
          if (ip>psize && ic>nsize)
          stop=1;
        }
        if (!leave) {
          if (i%B!=0)
            IOMem.BlockWrite(outbuf, (Extr_End+i)/B+2*m);
          oldp=psize;
          nsize=(nsize/B+1)*B-1;
          psize=(psize/B+1)*B-1;
          for(i=0; i<=nsize/B; i++) { /* Move to child node */
            IOMem.BlockRead(Extr_End/B+2*m+i, outbuf);
          IOMem.BlockWrite(outbuf, c*nodesize+i);
        }
        for (i=0; i<=psize/B; i++) { /* Move to parent */
          IOMem.BlockRead((nsize+1+Extr_End)/B+2*m+i, outbuf);
        if (p==-1) for (j=0; j<B; j++) IOMem[i*B+j]=IOMem[OUTBUF+j];
        else IOMem.BlockWrite(outbuf, p*nodesize+i);
      }
    }
    c=p; p=c/CPN-1;
    nsize=oldp;
    if (c==-1 || leave==1 || (p==-1 && maxi==-1))
      p=-2;
  }
}

void IOFlush() {
    signed long i, p;
```

```
        if (newleaf==-1) {
                for (i=0; i<CPN*B; i++)
                        IOMem[i]=IOMem[INS+i];
                maxi=CPN*B-1;
                newleaf=0;
        } else {
                /* Only rpn blocks are moved to the new rightmost node (not m) */
                for (i=0; i<rpn; i++)
                        IOMem.BlockWrite(ins+i, newleaf*nodesize+i);
                for (i=0; i<B; i++)     IOMem[OUTBUF+i]=-1;
                for (i=0; i<nodesize-rpn; i++) {
                        IOMem.BlockWrite(outbuf, newleaf*nodesize+rpn+i);
                }
                p=newleaf/CPN-1;
                if (p>-1)
                        IOSet_leaf(p, 0);                  /* Parent is no longer a leaf */
                IOSet_size(newleaf, ini);
                IOShift_Up_Insert(newleaf, ini);
                newleaf++;
        }
        ini=-1;
}

void IOInsertInternal(TYPE x)
{ signed long i, pos=ini;

        while ((pos>-1) && (IOMem[INS+pos]>x)) {
                IOMem[INS+pos+1]=IOMem[INS+pos];
                pos--;
        }
        IOMem[INS+pos+1]=x;
        ini++;

}

void IOInsert(TYPE x) {
        recs++;
        if (ini==RPN-1)
                IOFlush();
        IOInsertInternal(x);
}

void IOShift_Up_Melt(signed long c, signed long nsize) {
        signed long ip, ic, i, p;
        signed long psize;
        int stop, leave;

        p=c/CPN-1;
        while (p>-2) {
                psize=IOGet_size(p);
                if (nsize==-1 || psize==-1) {
                        printf("I/O-Shift_up_melt: indekser =-1");
                        exit(1);
                }
                if (p==-1)
                        IOMem.BlockRead(Extr_End/B +psize/B, 0);
                else
                        IOMem.BlockRead(p*nodesize, 0); /* parents first block */
                IOMem.BlockRead(c*nodesize+nsize/B, 1);  /* the childs last block */
                stop=0; leave=0;
```

```
		if (IOMem[B+nsize%B]<=IOMem[0])
			leave=stop=1;
		else IOMem.BlockRead(c*nodesize, 1);  /* the childs first block */
		/* Two-way merge using 3 blocks */
		i=ic=ip=0;
		while (!stop) {
			if ( (ip<=psize) && ( (ic>nsize) || (IOMem[ip%B]<IOMem[B+ic%B]) ) ) {
				IOMem[i%B+OUTBUF]=IOMem[ip%B]; ip++;
				if ((ip<=psize) && (ip%B==0)) {
					if (p==-1)
						IOMem.BlockRead((Extr_End+psize-ip+1)/B-1, 0);
					else IOMem.BlockRead(p*nodesize+ip/B, 0);
				}
			} else {
				IOMem[i%B+OUTBUF]=IOMem[B+ic%B]; ic++;
					if ((ic<=nsize) && (ic%B==0))
						IOMem.BlockRead(c*nodesize+ic/B, 1);
			}
			i++;
			if (i%B==0)
				IOMem.BlockWrite(outbuf, (Extr_End+i)/B+2*m-1);
			else if (i==nsize+1) {
				i=(nsize/B+1)*B;
				IOMem.BlockWrite(outbuf, (Extr_End+i)/B+2*m-1);
			}
			if (ip>psize && ic>nsize)
				stop=1;
		}
		if (!leave) {
			if (i%B!=0)
				IOMem.BlockWrite(outbuf, (Extr_End+i)/B+2*m);
			nsize=(nsize/B+1)*B-1;
			psize=(psize/B+1)*B-1;
			for(i=0; i<=nsize/B; i++) { /* Move to child node */
				IOMem.BlockRead(Extr_End/B+2*m+i, outbuf);
				IOMem.BlockWrite(outbuf, c*nodesize+i);
			}
			for (i=0; i<=psize/B; i++) { /* Move to parent */
				IOMem.BlockRead((nsize+1+Extr_End)/B+2*m+i, outbuf);
				if (p==-1) IOMem.BlockWrite(outbuf, (Extr_End+i+1)/B-1);
				else IOMem.BlockWrite(outbuf, p*nodesize+i);
			}
		}
		c=p;
		if ((c==-1) || leave==1) p=-2;
		else {
			p=c/CPN-1;
			nsize=IOGet_size(c);
		}
	}
}

void IOLeaf_melt(signed long p) {
	signed long c, c2, newsize, nsize, psize;


	psize=IOGet_size(p);
	if (psize==-1) {
		newleaf--;
		if (p!=newleaf) {
			for (c=0; c<nodesize; c++) {
				IOMem.BlockRead(newleaf*nodesize+c, outbuf);
```

```
                                IOMem.BlockWrite(outbuf, p*nodesize+c);
                        }
                        psize=IOGet_size(newleaf);
                        IOSet_size(p, psize);
                        IOShift_Up_Melt(p, psize);
                }
                IOSet_size(newleaf, -1);
                if (next_to_melt==p) next_to_melt=-1;
                if (next_to_melt==newleaf) next_to_melt=p;
        } else if ((next_to_melt==-1) || (next_to_melt==p)) {
                next_to_melt=p;
        } else {
                nsize=IOGet_size(next_to_melt);
                if (next_to_melt>p) {
                        c=next_to_melt; next_to_melt=p; p=c;
                        c=nsize; nsize=psize; psize=c;
                }
                for (c=0; c<(nsize+1)/B; c++)
                        IOMem.BlockRead(next_to_melt*nodesize+c, c);
                c2=c;
                for (c=0; c<(psize+1)/B; c++)
                        IOMem.BlockRead(p*nodesize+c, c2+c);
                c2+=c; c2*=B;
                IOMem.BlockRead(next_to_melt*nodesize+nsize/B, outbuf);
                for (c=0; c<(nsize+1)%B; c++)
                        IOMem[c2+c]=IOMem[OUTBUF+c];
                c2+=c;
                IOMem.BlockRead(p*nodesize+psize/B, outbuf);
                for (c=0; c<(psize+1)%B; c++)
                        IOMem[c2+c]=IOMem[OUTBUF+c];
                c2+=c;

                newsize=nsize+ psize+1;
                IOQuickSort(0, newsize);
                for (c=0; c<=newsize/B; c++)
                        IOMem.BlockWrite(c, next_to_melt*nodesize+c);
                IOSet_size(next_to_melt, (TYPE) newsize);
                IOShift_Up_Melt(next_to_melt, newsize);
                if (newsize>=RPN/2-1)
                        next_to_melt=-1;

                newleaf--;
                if (p!=newleaf) {
                        for (c=0; c<nodesize; c++) {
                                IOMem.BlockRead(newleaf*nodesize+c, outbuf);
                                IOMem.BlockWrite(outbuf, p*nodesize+c);
                        }
                        psize=IOGet_size(newleaf);
                        IOSet_size(p, psize);
                        IOShift_Up_Melt(p, psize);
                }
                IOSet_size(newleaf, -1);
        }
}

void IORefill(signed long p) {
        TYPE temp;
        int stop=0, lonely=0, no_childs=1;
        signed long i, c, max, inputs=CPN-1, psize;


        psize=IOGet_size(p);
```

```
/* Instaed of loading the childrens indices (are found in sizes): */
BlockReads++;
for (c=0; c<CPN; c++)   no_childs&=(sizes[(p+1)*CPN+c]==-1);
if (no_childs)
  if (p==-1) newleaf=0;
else { lonely=1; IOSet_leaf(p, 1); };

if (p!=-1 && lonely)
  IOLeaf_melt(p);
else {
  if (psize>-1) {
    for (c=0; c<=psize/B; c++)  /* Save eventual records in node */
      IOMem.BlockRead(p*nodesize+c, c);
    for (c=0; c<=psize; c++)    /* Move to end of workspace */
      IOMem[INS-1-c]=IOMem[psize-c];
    for (c=0; c<(psize+1)/B; c++)  /* Save the whole blocks */
      IOMem.BlockWrite(ins-1-c, Extr_End/B+m+c);
    for (c=0; c<B; c++)         /* Move the remainder to output buffer */
      IOMem[(outbuf+1)*B-1-c]=IOMem[INS-1-(psize/B)*B-c];
  }
  for (c=0; c<CPN; c++) {
    if (sizes[(p+1)*CPN+c]>-1)      /* and load last blocks of input */
      IOMem.BlockRead((p+1)*CPN*nodesize+c*nodesize+sizes[(p+1)*CPN+c]/B, c);
    else inputs-=1;
  }
  i=psize;
  while (!stop) {
    i++; max=0;
    while (sizes[(p+1)*CPN+max]==-1) max++;
    for (c=max; c<CPN; c++)  /* find index to max element */
    if (sizes[(p+1)*CPN+c]>-1 &&
        IOMem[c*B+sizes[(p+1)*CPN+c]%B]>IOMem[max*B+sizes[(p+1)*CPN+max]%B])
      max=c;
    temp=IOMem[max*B+sizes[(p+1)*CPN+max]%B];
    IOMem[(outbuf+1)*B-i%B-1]=temp; /* move max element to output block */
    if ((i+1)%B==0)
      if (p==-1) IOMem.BlockWrite(outbuf, (Extr_End+i+1)/B-1);
      else IOMem.BlockWrite(outbuf,       (Extr_End+i+1)/B+m-1);
    /* if input node empty or parent is full then stop. */
    if (sizes[(p+1)*CPN+max]==0) inputs-=1;
    if (i==RPN-1 || inputs==-1 ||
      (sizes[(p+1)*CPN+max]==0 && !IOGet_leaf(CPN*(p+1)+max))) stop=1;
    else if (sizes[(p+1)*CPN+max]>0 && sizes[(p+1)*CPN+max]%B==0) {
      IOMem.BlockRead((p+1)*CPN*nodesize+max*nodesize+(sizes[(p+1)*CPN+max]-1)/B, max);
  }
  --sizes[(p+1)*CPN+max];
};
IOSet_size(p, i);
psize=(TYPE) i;
if ((i+1)%B!=0) {  /* adjust i to nearest whole multiplum of B etc */
  i=(i/B+1)*B-1;
  if (p==-1) IOMem.BlockWrite(outbuf, (Extr_End+i+1)/B-1);
  else IOMem.BlockWrite(outbuf, (Extr_End+i+1)/B+m-1);
}
if (p>-1) {
  for (c=0; c<=i/B; c++)  /* Move records to node */
    IOMem.BlockRead(Extr_End/B+c+m, (i+1)/B-c-1);
  if ((i-psize)>0)
    for (c=0; c<=psize; c++)
      IOMem[c]=IOMem[c+(i-psize)];
  for (c=0; c<=i/B; c++)  /* Move records to node */
    IOMem.BlockWrite(c, p*nodesize+c);
```

```
        }
        for (c=CPN-1; c>=0; c--)    /* Refill underfull children */
          if ( ((p+1)*CPN+c<newleaf) && (sizes[(p+1)*CPN+c]<RPN/2-1) )
            IORefill(CPN*(p+1)+c);
        if (p==-1) {
          for (c=0; c<=i/B; c++)    /* Load root records */
            IOMem.BlockRead(Extr_End/B+c, (i+1)/B-c-1);
          if ((i-psize)>0)
            for (c=0; c<=psize; c++)
              IOMem[c]=IOMem[c+(i-psize)];
        }
        lonely=1;
        for (c=0; c<CPN; c++)   lonely&=(sizes[(p+1)*CPN+c]==-1);
        if (p>-1 && lonely)     IOSet_leaf(p, 1);
      }
}

TYPE IODelete_max() {
        TYPE max;

        if (recs==-1) {
                printf("Delete-max on IO-queue");       exit(1);
        }
        if (newleaf>0 && maxi==-1)
                IORefill(-1);
        if ((maxi<0) || ((ini>-1) && (IOMem[maxi]<IOMem[INS+ini]))) {
                max=IOMem[INS+ini--];
        } else {
                max=IOMem[maxi--];
        }
        recs--;
        return max;
}

void IOInit() {
        signed long i;

        IOMem.Init();
        maxi=ini=recs=-1;
        newleaf=next_to_melt=-1;
        for (i=INDS; i<OUTBUF; i++)
                IOMem[i]=-1;
        for (i=0; i<2*(N/B); i++) {
                leaf[i]=1; sizes[i]=-1;
        }
}




/*************************************************************/
/****                   Main                             ***/
/*************************************************************/

void WriteFile(char *Name) {
  FILE *out;
  out = fopen(Name,"a+");
  fprintf(out,"%d : %ld : %ld \n",alpha,N,BlockReads+BlockWrites);
  fclose(out);
}
```

```
void RunBTree() {
  long a;
  printf("BTree \n");
  InitBTree();
  for(a = 0; a < N; a++)
    BTreeInsert(StartData[a]);
  BTreeMem.Init();
  for(a = 0; a < N; a++) {
    if (Operations[a] < alpha)
      BTreeInsert(OpData[a]);
    else
      BTreeDeleteMax();
    }
  WriteFile("priBTree.out");
}

void RunBHeap() {
  long a;
  printf("B-Heap \n");
  HeapInit();
  for(a = 0; a < N; a++)
    HeapInsert(StartData[a]);
  HeapMem.Init();
  for(a = 0; a < N; a++) {
    if (Operations[a] < alpha)
      HeapInsert(OpData[a]);
    else
      HeapDelete_max();
    }
  WriteFile("priBHeap.out");
}


void RunIOQueue() {
  long a;
  printf("IO-queue \n");
  IOInit();
  for(a = 0; a < N; a++)
    IOInsert(StartData[a]);
  IOMem.Init();
  for(a = 0; a < N; a++) {
    if (Operations[a] < alpha)
      IOInsert(OpData[a]);
    else
      IODelete_max();
    }
  WriteFile("priIOqueue.out");
}


void RUN(long qq) {
  N = qq;

  printf("*****   alpha = %ld     N = %ld         ****\n",(long)alpha,N);

//  RunBTree();
//  RunBHeap();
  RunIOQueue();
}

void main () {
  long qq;
```

```
  randomize;

  if (BlockSize % 2 != 0)  {
    printf("Blocksize must be even \n");
    exit(42);
    }

  printf("We begin\n");
  MakeStartData();

  alpha = 25;
  for(qq = 10000; qq <= 100000; qq+=10000)
    RUN(qq);

  alpha = 50;
  for(qq = 10000; qq <= 100000; qq+=10000)
    RUN(qq);

  alpha = 75;
  for(qq = 10000; qq <= 100000; qq+=10000)
    RUN(qq);
}
```

Bibliography

[1] G.M. Adel'son-Velski and E.M. Landis, An algorithm for the organization of information, *Soviet Mathematics Doklady* **3** (1962) 1259-1263

[2] A. Aggarwal, B. Alpern, A.K. Chandra and M. Snir, A model for hierachical memory, *Proc. of the 19th Anual ACM Symposium on the Theory of Computing*, ACM, New York (1987) 305-314

[3] A. Aggarwal, A.K. Chandra and M. Snir, Hierachical memory with block transfer, *Proceedings of the 28th Anual Symposiom on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos (1987) 204-216

[4] A. Aggarwal and J.S. Vitter, The input/output complexity of sorting and related problems, *Communications of the ACM* **31** (1988) 1116-1127

[5] L. Arge, M. Knudsen and K. Larsen, A general lower bound on the I/O-complexity of comparisons-based algorithms, *Workshop on Algorithms and Data Structures 1993*

[6] L. Arge, External-Storage Data Structures for Plane-Sweep Algorithms, BRICS Report Series **RS-94-16**, Department of Computer Science, University of Aarhus

[7] L. Arge and J.S. Vitter, Optimal Dynamic Interval Management in External-Memory, *Foundations on Computer Science 1996*

[8] R.D. Barve, E.F. Grove and J.S. Vitter, Simple randomized mergesort on parallel disks, *Proceedings on the 8th Anual ACM Symposium on Parallel Algorithms and Architectures (SPAA 96)*, Padua Italy 1996

[9] R. Bayer and E. McCreight, Organization and maintance of large order indexes, *Acta Informatica* **1** (1972) 173-189

[10] S. Carlsson, Heaps, Ph.D. Thesis, Department of Computer Sciences, Lund University, Lund, Sweden (1986)

[11] Y. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff and J.S. Vitter, External-memory graph algorithms, *Proceedings of the Sixth (ACM)-(SIAM) Annual Symposium on Discrite Algorithms* (1995) 139-149

[12] Y. Chiang, Dynamic and I/O-efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results, Ph.D. Dissertation, Department of Computer Science, Brown University Providence, Rhode Island 02912 **CS-95-27** (1995)

[13] F.Y. Chin, J. Lam and I. Chen, Efficient parallel algorithms for some graph problems, *Communications of the ACM* **vol 25** (1982)

[14] D. Comer, The ubiquitous B-tree, *ACM computing surveys* **vol 11** (1979) 121-137

[15] T.H. Cormen, Fast permuting on disk arrays, *Journal on Parallel and Distributed Computing* **17** (1993) 41-57

[16] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduktion to algorithms*, MIT press/McGrae-Hill (1994)

[17] R. Decker, *Data structures*, Prentice-Hall (1989)

[18] P.J. Denning, The working set model for program behavior, *Communications of the ACM* **vol 13** (1968) 323-333

[19] W.R. Dufrene Jr and F.C. Lin, An efficient external sort algorithm with no additional space, *The Computer Journal* **vol 35** (1992) 308-310

[20] E. Feuerstein and A. Marchetti-Spaccamela, Memory paging for connectivity and path problems in graphs, *International Symposium on Algorithms and Computing 1993*, Lecture Notes on Computer Science **762**, Springer Verlag Berlin 416-425

[21] P. Flajolet and J.M. Steyaert, A branching process arising in dynamic hashing, trie searching and polynomial factorization, *Nineth International Colloguium on Automata, Languages and Programming 1982*, Lecture notes on computer science **140**, Springer Verlag Berlin 239-251

[22] A.V. Goldberg, Optimization algorithms for large networks, *European Symposium on Algorithms 1995*, Lecture notes on computer science **855**, Springer Verlag Berlin 1-9

[23] G.H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley 1991

[24] G. H. Gonnet and P. Larson, External hashing with limited internal storage, *Journal of the ACM* **Vol 35** (1988) 161-184

[25] H.N. Graham and O. Hell, On the history of the minimum spanning tree problem, *Ann. Hist. Comput.* **7** (1985) 43-57

[26] F.M. auf der Heide, Dynamic hashing strategies, *Mathimatical Foundations of Computer Science 1990*, Lecture Notes on Computes Science **452** Springer Verlag Berlin (1990) 76-87

[27] S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Informatica* **vol 17** (1982) 157-184

[28] G.A. Hyslop and E.A. Lamagna, Performance of distributive partitioned sort in a demand paging enviroment, *Information Procesing Letters* **vol 25** (1987) 61-64

[29] B.H.H. Juurlink and H.A.G. Wijshoff, The parallel hierarchical memory model, *Scandinavian Workshop on Algorithm Theory 1994*, Lecture Notes on Computer Science **824** Springer Verlag Berlin 240-251

[30] D.E. Knuth, *The art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley (1973)

[31] E. Liljedahl and O.K. Christensen, *Grafalgoritmer og kompleksitet: Datalogi 1P noter 1991/92*, *diku*tryk*, Datalogical Institute Copenhagen University (1992)

[32] B.M.E. Moret and H.D. Shapiro, An empirical analysis of algorithms for constructing a minimum spanning tree, *2.nd workshop WADS'91*, Lecture Notes on Computer Science **519** Springer Verlag Berlin 400-411

[33] K. Mehlhorn, *Data structures and algorithms*, Vol. 1: *Sorting and Searching*, Springer Verlag Berlin (1984)

[34] M.H. Nodine, M.T. Goodrich and J.S. Vitter, Blocking for external graph search, *Proceedings of the ACM SIGACT-SIGMOD-SIGART symposium on Priciples Of Database Systems* (1993)

[35] M.H. Nodine and J.S. Vitter, Paradigms for optimal sorting with multiple disks, *Proc. on the 26th Hawaii international conference on systems science*, IEEE (1993)

[36] J.F. Sibeyn and T. Harris, Exploiting locality in LT-RAM computations, *Scandinavian Workshop on Algorithms Theory 1994*, Lecture Notes on Computer Science **824** Springer Verlag Berlin 338-349

[37] A. Silberschatz and J.L. Peterson, *Operating System Concepts*, Addison-Wesley (1988)

[38] M.R. Spiegel, *Mathematical handbook of formulas and tables*, McGraw-Hill (1968)

[39] R.E. Tarjan, *Data structures and network algorithms*, SIAM (1983)

[40] R.E. Tarjan, Amortized computational complexity, *SIAM Journal on Algoritms and Discrete Methods* **vol 6** (1985)

[41] A.I. Verkamo, Performance of quicksort adapted for virtual memory use, *The Computer Journal* **vol 30** (1987) 362-371

[42] A.I. Verkamo, Sorting in hierarichal memories, Department of Computer Science, University of Helsinki **A-1988-1** (1988)

[43] L.T. Wegner and J. Teuhola, The External Heapsort, *IEEE Transactions on software engineering* **Vol 15** (1989)

[44] J.S. Vitter and E.A.M. Shriver, Algorithms for parallel memory I: Two level memories, *Algorithmica* **12** (1994) 110-147

[45] J.S. Vitter and E.A.M. Shriver, Algorithms for parallel memory II: Hierarchical Multilevel Memories, *Algorithmica* **12** (1994) 148-169