Theoretical and practical efficiency of priority queues

Claus Jensen

M. Sc. Thesis

Department of Computer Science
University of Copenhagen, Denmark

May 2006

Abstract

This is a study of the theoretical and practical efficiency of priority queues. The priority queue is an old and well studied data structure on which a great deal of theoretical and practical work has already been done. A priority queue can be realised in many ways, some of the most commonly used data structures for this are binary heaps and binomial queues.

In this thesis the efficiency of priority queues is studied in three separate papers where one is focusing on the theoretical efficiency of priority queues and the other two papers focus on the practical efficiency.

In the first paper studying the practical efficiency of priority queues (An extended truth about heaps) a number of alternative implementations of in-place d-ary heaps are studied. An experimental evaluation of the implementations and the C++ standard library heap are performed. The implementations are evaluated using different types of inputs and ordering functions. The results of the experimental evaluation show that no single heapifying strategy has the best performance for all the different types of inputs and ordering functions, but that bottom-up heapifying has a good performance for most types of inputs and ordering functions.

In the second paper studying the practical efficiency of priority queues (An experimental evaluation of navigation piles) three different implementations of navigation pile are studied. The efficiency of the implementations is experimentally evaluated together with two implementations of binary heaps, again different types of inputs and ordering functions have been used. Furthermore, experiments using several different operation-generation models are performed. The experiments show that when element moves are expensive then navigation piles can be an alternative to binary heaps. In addition to the study of the practical efficiency of navigation piles, the first-ancestor technique and a new and simpler way to make static navigation piles dynamic are introduced.

In the third paper (A framework for speeding up priority-queue operations) the theoretical efficiency of priority queues is studied, and a framework for reducing the number of element comparisons performed in priority-queue operations is introduced. The framework gives a priority queue which guarantees the worst-case cost of $O(1)$ per *find-min* and *insert*, and the worst-case cost of $O(\log n)$ with at most $\log n + O(1)$ element comparisons per *delete-min* and *delete*. Here, n denotes the number of elements stored in the data structure prior to the operation in question, and $\log n$ equals $\max\{1, \log_2 n\}$. Furthermore, in addition to the above-mentioned operations, a priority queue that provides *decrease* (also called *decrease-key*) is given. This priority queue achieves the worst-case cost of $O(1)$ per *find-min*, *insert*, and *decrease*; and the worst-case cost of $O(\log n)$ with at most $\log n+O(\log\log n)$ element comparisons per *delete-min* and *delete*.

Preface

This collection of papers is the M. Sc. thesis of Claus Jensen. The thesis is written at the Department of Computer Science, University of Copenhagen, under the supervision of Jyrki Katajainen.

The three papers that constitute the main substance of this M. Sc. thesis are based on the following technical reports (which are all available through the CPH STL website [Dep06]):

Claus Jensen, Jyrki Katajainen, and Fabio Vitale. An extended truth about heaps. CPH STL Report 2003-5.

Claus Jensen and Jyrki Katajainen. An experimental evaluation of navigation piles. CPH STL Report 2006-3.

Amr Elmasry, Claus Jensen, and Jyrki Katajainen. A framework for speeding up priority-queue operations. CPH STL Report 2004-3.

I would like to thank my supervisor Jyrki Katajainen for introducing me to the world of scientific research and teaching me how to write research papers. I would also like to thank the two other co-authors of the papers, Amr Elmasry and Fabio Vitale, for their contributions.

Contents

Chapter 1

Introduction

A priority queue is a data structure that maintains a set of elements, each element is associated with a key [CLRS01]. The following set of operations is supported by a min priority-queue Q:

find-min(Q). Returns a pointer to a node containing a minimum element of the priority queue Q.

insert(Q, x). Inserts node x, which has already been constructed to contain an element, into priority queue Q.

delete-min(Q). Removes a minimum element and the node, in which it is contained, from priority queue Q.

In some cases the following operations are also supported:

delete(Q, x). Removes the node x, and the element it contains, from priority queue Q.

decrease(Q, x, e). Replaces the element at node x with element e. It is assumed that x is in Q and that e is no greater than the element earlier stored at x.

In the papers "An extended truth about heaps" and "An experimental evaluation of navigation piles" the data structures investigated support the operations *find-min*, *insert*, and *delete-min*, however, since we compare our implementations to the priority queue of the C++ standard library [Bri03] we choose to follow their naming convention. Using the the C++ standard library naming convention *find-min* is called *top*, *insert* is called *push*, and *delete-min* is called *pop*.

In Chapter 2 the paper "An extended truth about heaps" describes the experimental evaluation of various heap variants. The heap variants are grouped into two heapifying strategies top-down and bottom-up. The following top-down heapifying strategies has been implemented: Basic [Joh75, Oko80, Wil64] and one-sided binary search [GZ96]. Also, the following bottom-up heapifying strategies have been implemented: Basic [Knu98, §5.2.3, exercise 18], two levels at a time (folklore), binary search [Car87, GM86], exponential binary search [Pas], and move saving [Weg93]. Furthermore, five different *push* strategies have been implemented. One of the goals of the study was to look for an approach that performs best for different types of inputs and ordering functions therefore the experiments used the following types of input parameters: 1) built-in unsigned integers; 2) bigints, as described in the book of Bulka and Mayhew [BM00], where unsigned integers are represented as a string of digits; 3) built-in unsigned integers combined with an ordering function that computes the natural logarithm of the elements before comparing them; and 4) pairs of unsigned integers and bigints and an ordering function that computes the natural logarithm function for the unsigned integers in element comparisons.

In Chapter 3 the paper "An experimental evaluation of navigation piles" describe the experimental evaluation of three different navigation pile implementations. The first implementation is based on the original proposal given in the paper on navigation piles [KV03]. The elements in this implementation are stored in a resizable array and above this array a bit container is constructed storing a packed form of the navigation information. In the second implementation the bit container is replaced with an index array. In the third implementation the whole data structure is realized using concrete nodes and pointers. Again different types of inputs and ordering functions are used in the experimental evaluation, to be more exact the first three of the aforementioned input parameters are used.

In Chapter 4 "A framework for speeding up priority-queue operations" the following is presented: In Section 4.4, a structure called a *two-tier binomial queue* is given, it guarantees the worst-case cost of $O(1)$ per *find-min* and *insert*, and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log\log n)$ element comparisons per *delete-min* and *delete*. In Section 4.5, a priority queue called a *multipartite binomial queue* is described, it achieves the bound of at most $\log n+O(1)$ element comparisons per *delete-min* and *delete*. Both of these data structures have binomial trees as there basic building blocks. In Section 4.6, an application of the framework is described, by using a multipartite binomial queue in adaptive heapsort [LP93], a sorting algorithm is obtained that is optimally adaptive with respect to the inversion measure of disorder, and which sorts a sequence having n elements and I inversions with at most $n\log(I/n) + O(n)$ element comparisons. In Section 4.7, a priority queue, called a *multipartite relaxed binomial queue* is presented, in addition to the above-mentioned operations it also provides *decrease*. This priority queue uses run-relaxed binomial trees [DGST88] as its basic building blocks. A multipartite relaxed binomial queue guarantees the worst-case cost of $O(1)$ per *insert*, *find-min*, and *decrease*; and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log\log n)$ element comparisons per *delete-min* and *delete*.

Chapter 2

An extended truth about heaps

Abstract. We describe a number of alternative implementations for the heap functions, which are part of the C++ standard library, and provide a through experimental evaluation of their performance. In our benchmarking framework the heap functions are implemented using the same set of utility functions, the utility functions using the same set of policy functions, and for each implementation alternative only the utility functions need be modified. This way the programs become homogeneous and the underlying methods can be compared fairly.

Our benchmarks show that the conflicting results in earlier experimental studies are mainly due to test arrangements. No heapifying approach is universally the best for all kinds of inputs and ordering functions, but the bottom-up heapifying performs well for most kinds of inputs and ordering functions. We examine several approaches that improve the worst-case performance and make the heap functions even more trustworthy.

2.1 Introduction

Context. The Standard Template Library (STL) is an integrated part of the standard library for the C++ programming language [Bri03]. This work is part of the CPH STL project where the goal is to:

- study and analyse existing specifications for and implementations of the STL to determine the best approaches to optimization,
- provide an enhanced edition of the STL and make it freely available on the Internet,
- provide cross-platform benchmark results to give library users a better basis for assessing the quality of different STL components,

- develop software tools that can be used in the development of component libraries, and
- carry out experimental algorithmic research.

For further information about the project, see the CPH STL website [Dep06].

Goal of this study. Earlier studies by LaMarca and Ladner [LL96, LL99] pointed out that the memory references are more local for 4-ary and 8-ary heaps than for binary heaps, which give better cache behaviour and faster programs. On the other hand, a later study by Sanders [San00] showed that in some cases binary heaps are faster than 4-ary heaps. Therefore, the goal set for this study was

1. to produce a framework for measuring the efficiency of various heap variants,
2. to repeat the tests of earlier studies with our framework on contemporary computers, and
3. to seek an approach that performs best for different kinds of inputs and ordering functions.

Preliminary definitions. For an integer $d \geq 2$, a ***d-ary heap*** — invented by Williams [Wil64] for $d = 2$ and generalized for $d > 2$ by Johnson [Joh75] — with respect to a given ordering function $less()$ is a sequence of elements with the following properties:

Shape: Internally, it is a nearly complete (or left-complete) d-ary tree.

Capacity: Each node of that tree stores one element.

Order: For each ***branch*** of the tree, i.e. for a node having at least one child, the element y stored at that node should be no smaller than the element x stored at any child of that node; or stated in another way, $less(y, x)$ must return false.

Representation: The elements in the tree are stored in breath-first order.

Informally, such a heap is called a d-ary ***max-heap***. We assume that the reader is familiar with the basic concepts related to heaps as described, for example, in [CLRS01, chap. 6].

An ***iterator*** is a generalization of a pointer that indicates a position of an element, provides operations for accessing the elements stored and operations for moving to neighbouring positions. All the iterators considered in this paper are assumed to be random-access iterators (for a precise definition of this concept, see [Bri03, §24.1]). For example, if A and Z are random-access iterators and i is a nonnegative integer, $Z{-}A$, $\texttt{++}A$, $\texttt{--}Z$, $A{+}i$, $Z{-}i$, $A[i]$, and $*A$ are all allowable expressions having the same meaning as the corresponding pointer expressions. For iterators A and Z, we use $[A \mathinner{..} Z)$ to denote the ***sequence*** of ***positions*** $A, A{+}1, \ldots, A{+}\big((Z{-}A){-}1\big)$ storing the elements $A[0], A[1], \ldots, A[(Z{-}A){-}1]$.

According to the C++ standard [Bri03, §25.3.6] every realization of the standard library must provide the following ***heap functions***:

template <**typename** position, **typename** ordering>
void *push_heap*(position A, position Z, ordering *less*);
Requirement: $[A\,..\,Z–1)$ stores a heap with respect to ordering function *less*().

Effect: Make the sequence stored in $[A\,..\,Z)$ into a heap.

template <**typename** position, **typename** ordering>
void *pop_heap*(position A, position Z, ordering *less*);
Requirement: $[A\,..\,Z)$ stores a heap with respect to ordering function *less*().

Effect: Swap the element stored at position A with the element stored at position $Z-1$, and make the sequence stored in $[A\,..\,Z-1)$ into a heap.

template <**typename** position, **typename** ordering>
void *make_heap*(position A, position Z, ordering *less*);
Effect: Convert the sequence stored in $[A\,..\,Z)$ into a heap with respect to ordering function *less*().

template <**typename** position, **typename** ordering>
void *sort_heap*(position A, position Z, ordering *less*);
Requirement: $[A\,..\,Z)$ stores a heap with respect to ordering function *less*().

Effect: Sort the sequence stored in $[A\,..\,Z)$ with respect to ordering *less*().

When these functions are available, it is easy to implement a priority queue class (see [Bri03, §23.2.3.2]) relying on any sequence supporting expansion and shrinkage at the back end.

Our work versus earlier work. The C++ standard [Bri03, §25.3.6] gives tight complexity requirements for the heap functions. Letting n denote the size of the sequence manipulated, *push_heap*() should run in $O(\log_2 n)$ time and perform at most $\log_2 n$ element comparisons, *pop_heap*() should run in $O(\log_2 n)$ time and perform at most $2\log_2 n$ element comparisons, *make_heap*() should perform at most $3n$ element comparisons, and *sort_heap*() at most $n\log_2 n$ element comparisons. Moreover, the intention is that the heap functions are memoryless. That is, no extra information is passed from one invocation to another, but temporary storage within a single function can be used.

These requirements were the main reason why we had to reject many of the proposals presented in the literature. Some methods only guarantee good amortized bounds, e.g. sequence heaps of Sanders [San00]; some methods have too large constant factors in their worst-case complexity bounds, e.g. external heaps of Wegner and Teuhola [WT89]; and some methods require extra space, e.g. navigation piles of Katajainen and Vitale [KV03]. Also, in the current implementation of the CPH STL, done by Jensen [Jen01], the heap functions based on 8-ary heaps do not fulfil the requirements, and in our preliminary experiments they turned out to be slow if element comparisons are expensive.

As far as we know, the heap functions based on 2-ary, 3-ary, and 4-ary heaps are the only ones that fulfil the requirements laid down in the C++ standard, except that for the complexity of the *sort_heap*() function. We describe a number of alternative implementations for the heap functions and provide a framework for benchmarking their performance. We apply the policy-based approach advocated by Alexandrescu [Ale01a, Ale01b], as well as borrow elements from the earlier work done in our research group [Boj98, BKS00, Jen01].

In the recent experimental studies on the efficiency of priority queues, the elements used as input were relatively small. LaMarca and Ladner [LL96, LL99] used 32-bit and 64-bit integers, and Sanders [San00] 32-bit integer keys associated with 32-bit satellite data. The focus in these studies was on cache behaviour and, when the elements manipulated are small, one can see dramatic cache effects. On the other hand, in the experimental studies on sorting it is a tradition to consider larger elements: in industry-strength benchmarks 100-byte records with 10-byte keys are used (see, e.g. [NBC$^+$95]). In the experiments of Edelkamp and Stiegeler [ES02] expensive ordering functions were considered.

For the heap functions the element type (derived from the position type) and the type of the ordering function are given as templete parameters. That is, these library functions should perform well for all kinds of elements and ordering functions. As a first approximation of this, we test all methods with four different kinds of input:

1. Built-in unsigned ints, for which both element comparisons and element moves are cheap.

2. Bigints that represent an unsigned integer as a string of its digits (chars). The bigint class used by us is described in the book by Bulka and Mayhew [BM00, chap. 12]. For bigints element comparisons are relatively cheap, whereas element moves are expensive.

3. Unsigned ints with an ordering function that computes the natural logarithm of the given numbers before comparing them. This is just one of the expensive ordering functions considered by Edelkamp and Stiegeler [ES02].

4. Pairs of unsigned ints and bigints applying the natural logarithm function for unsigned ints in element comparisons. This makes both element comparisons and element moves expensive.

To sum up, the task of a library implementer is much harder than that of performance engineer.

Contents. We start by reviewing the implementation alternatives for the heap functions. Our focus is on the functions *push_heap*() and *pop_heap*(). Thereafter, we describe our benchmarking framework for comparing the alternatives presented. Finally, we report the results of our benchmarks.

Table 2.1: The worst-case performance of various versions of *push_heap*(). If the heap stores $n \geq 1$ elements, the height h is equal to $\lfloor \log_d \big((d-1)(n-1)+1\big) \rfloor$.

approach	reference	# comparisons	# moves
bottom-up			
– basic	[Joh75, Wil64]	$h+1$	$h+2$
– two levels at a time	folklore	$\lfloor h/2 \rfloor + 1$	$h+2$
– binary search	[Car87, GM86]	$\lceil \log_2(h+1) \rceil$	$h+2$
– exponential binary search	[Pas]	$2\lfloor \log_2 h \rfloor + 1$	$h+2$

2.2 Implementation alternatives

Function *push_heap*(). Assume that a heap is stored in $[A\,..\,Z-1)$ and that a new element in $Z-1$ is to be inserted into that heap. Consider the path upwards from the new last leaf $(Z-1)$ to the root (A), we call it the ***siftup path***. Given the path, the task is to insert the new element $*(Z-1)$ into the sorted sequence of elements stored on the siftup path.

According to the original proposals by Williams [Wil64] ans Johnson [Joh75] the insertion is done by a simple linear scan starting from the new last leaf. One could reduce the number of element comparisons — keeping the number of element moves unchanged — by performing the test whether to stop the traversal only at every second level and backtrack if one goes too far up. Also, since the elements stored on the siftup path are in sorted order, binary search could be used to determine the final destination of the new element [Car87, GM86]. Yet another alternative, as communicated to us by Pasanen [Pas], is to use exponential binary search [BY76] instead of binary search.

In Table 2.1 we summarize the worst-case performance of the implementation alternatives mentioned.

Function *pop_heap*()**.** Consider a heap stored in $[A\,..\,Z)$. Assume that we have put element $*(Z-1)$ from the last leaf aside and moved element $*A$ from the root to the last leaf so that we have a hole at the root. Now the task is to embed the element put aside into the sequence stored in $[A\,..\,Z-1)$ and remake it into a heap. To carry out this task we consider the special path starting from the root, ending at a leaf, and going down at each level to the child which stores the maximum element among all the children. We call this the ***siftdown path***. The elements on the siftdown path appear in sorted order. In addition to finding the path, the task is to insert the element put aside into this sorted sequence.

In the ***top-down heapifying***, used in the original articles by Williams [Wil64] and Johnson [Joh75], the siftdown path is traversed down as long as the node under consideration is not a leaf and the element stored at that node is larger than the element put aside. Simultaneously, the elements met are moved one level up to get the hole down. When the traversal stops, the element put aside is stored at the hole. If the arity of the tree is d, at most d element comparisons are done at each level: at most $d-1$ to find the maximum element stored at the children and one to determine whether we should stop or not.

We could reduce the number of element comparisons by carrying out the test whether we should stop only at every second level, and backtracking if we proceed too far down. Another possibility, proposed by Gu and Zhu [GZ96], is to calculate the height of the tree in the beginning, do the stop test first at level $\lfloor h/2 \rfloor$, and backtrack with linear search or repeat the process recursively downwards. That is, a one-sided binary search is performed on the sequence stored on the siftdown path. As shown in [Car92], it is possible to reduce the number of element comparisons even further, but this improvement is only of theoretical interest.

In the ***bottom-up heapifying***, proposed by Floyd (as cited in [Knu98, §5.2.3, exercise 18]), the siftdown path is first traversed down until a leaf is met, the elements on the path are moved one level up, and thereafter another traversal up the tree along the siftdown path is done to determine the final destination of the element put aside, by moving the elements met along. When traversing down at most $d-1$ element comparisons are done at each level, and when traversing up one element comparison is done at each level considered. The point is that often only a few nodes are visited during the upwards traversal. As to the number of element comparisons, the worst case is the same as that for the basic top-down version, but the number of element moves can be almost twice as high.

The number of element comparisons performed in the worst case can be reduced in several ways depending on how the upwards traversal is done. First, the test whether to stop or not could be done only at every second level. Second, the final destination of the element put aside could be determined using binary search [Car87, GM86]. Third, the final destination could be determined using exponential binary search [Pas]. Fourth, one could save element moves by omitting the moves during the downwards traversal and doing them first after the final destination of the new element is known. In the original proposal by Wegener [Weg93] the upwards traversal was done by linear search, but any of the above-mentioned approaches could be used.

The worst-case performance of all the implementation alternatives mentioned is summed up in Table 2.2.

Function *make_heap*()**.** In the paper by Bojesen et al. [BKS00] various heap construction methods were rigorously analysed both theoretically and experimentally. The conclusion was that the standard heap construction method [Flo64] performs almost optimally in all respects if the nodes are handled in depth-first order. Our experiments show that the extreme code-tuning described in [BKS00] is not necessary since on contemporary computers the few instructions saved can be executed in pipeline or even in parallel with other interdependent instructions. According to Okoma [Oko80], for a d-ary heap containing n elements, the heap construction requires at most $dn/(d–1)$ element comparisons.

Function *sort_heap*()**.** It is well-known that heapsort cannot compete against quicksort or mergesort. Moreover, since quicksort does not have a good worst-case performance guarantee, we rely on mergesort in the realization of this function. Even if temporary storage could be allocated, we decided to avoid that — to make the function more robust — and use the in-place mergesort algorithm

Table 2.2: The worst-case performance of various versions of *pop_heap*(). If the heap stores $n \geq 1$ elements, the height h is equal to $\lfloor \log_d ((d-1)(n-1)+1) \rfloor$.

approach	reference	# comparisons	# moves
top-down			
– basic	[Joh75, Oko80, Wil64]	dh	$h+2$
– two levels at a time	folklore	$(d-1)h+\lfloor h/2 \rfloor+1$	$h+2$
– one-sided binary search	[GZ96]	$(d-1)h+\lfloor h/2 \rfloor+1$	$h+2$
bottom-up			
– basic	[Knu98, §5.2.3, exercise 18]	dh	$2h+2$
– two levels at a time	folklore	$(d-1)h+\lfloor h/2 \rfloor+1$	$2h+2$
– binary search	[Car87, GM86]	$(d-1)h+\lceil \log_2(h+1) \rceil$	$2h+2$
– exponential binary search	[Pas]	$(d-1)h+2\lfloor \log_2 h \rfloor+1$	$2h+2$
– move saving	[Weg93]	dh	$h+2$

developed by Katajainen et al. [KPT96].

It is interesting to note that in the experiments reported in the original paper by Katajainen et al. [KPT96] heapsort was a clear winner compared to in-place mergesort. Contemporary computers favour local memory accesses and most memory accesses in in-place mergesort are such, which makes it a noteworthy alternative. Experiments carried out in our research group by Sloth et al. [SLK03] showed that for uniformly distributed integer data in-place mergesort is almost as fast as introsort [Mus97], which is an adaptation of median-of-three quicksort. We want to point out that, even its practical utility, the implemented variant of in-place mergesort requires $n \log_2 n + O(n)$ element comparisons, so some fine-tuning is necessary to reach the bound $n \log_2 n$ set down in the C++ standard.

2.3 Benchmarking framework

Heap policies. Our policy-based implementation of the heap functions, when the underlying data structure is an in-place d-ary heap, has two parts: policy functions and utility functions. A ***policy*** means a minimal set of core functions that are enough to implement all the utility functions. Given the policy functions and utility functions it is then possible to implement all the heap functions.

It is natural to represent a policy as a class which contains all the core functions. The declaration of such a class is given in Figure 2.1. Most member functions are **const** functions since they do not change the state of a policy object. The only information which is stored in a policy object is n, the current number of elements in the heap.

The meaning of most member functions should be clear from their name, except that of the functions *top_all_present*() and *top_some_absent*() which compute for a given node the index of the child storing the maximum element among all the children. Internally, the two functions are similar — a linear scan over the children is performed and the index of the node storing the maximum ele-

```
#include <iterator> // defines std::iterator_traits

template <int d, typename position, typename ordering>
class heap_policy {
public:
  typedef typename std::iterator_traits<position>::difference_type index;
  typedef typename std::iterator_traits<position>::difference_type level;
  typedef typename std::iterator_traits<position>::value_type element;

  heap_policy(index n = 0);
  bool is_root(index) const;
  bool is_first_child(index) const;
  index size() const;
  level depth(index) const;
  index root() const;
  index leftmost_node(level) const;
  index last_leaf() const;
  index first_child(index) const;
  index parent(index) const;
  index ancestor(index, level, level) const;
  index top_all_present(position, index, ordering) const;
  index top_some_absent(position, index, ordering) const;
  void update(position, index, const element&);
  void erase_last_leaf(position, ordering);
  void insert_new_leaf(position, ordering);

private:
  index n;
};
```

Figure 2.1: Declaration of the heap-policy class.

ment is recalled. To make an efficient loop unrolling possible, we separated the case where the branch considered has all its d children, and the case where some of the children is missing.

The general heap-policy class is specialized for specific values of the arity d. The challenge is to implement the functions *depth*(), *leftmost_node*(), and *ancestor*() efficiently. For a node with index i and level λ, these can be calculated using the formulas:

$\mathit{depth}(i)$: $\lfloor \log_d \left((d-1)i+1\right) \rfloor$

$\mathit{leftmost\_node}(\lambda)$: $(d^{\lambda}-1)/(d-1)$

$\mathit{ancestor}(i, \lambda, \Delta)$: $\lfloor (i-\mathit{leftmost\_node}(\lambda))\,/d^{\Delta} \rfloor + \mathit{leftmost\_node}(\lambda - \Delta)$, where $\Delta \in \{0, 1, \ldots, \lambda\}$.

In the implementation of these functions we used two precomputed tables, one giving the index of the leftmost node at each level and another giving the powers of d for $d = 3$. To compute the depth, for $d = 2$ and $d = 4$ the math library function *ilogb*() was used, and for $d = 3$ binary search over the first precomputed table was performed.

Utility functions. As in many textbooks, we implement the heap functions using two utility functions *siftup*() and *siftdown*():

template <**typename** position, **typename** index, **typename** element,
typename ordering, **typename** policy>
void *siftup*(position A, index i, element x, ordering *less*, policy& p);
Requirement: $A[i]$ contains a hole.

Effect: Insert element x into the sorted sequence stored on the siftup path staring from the node with index i.

template <**typename** position, **typename** index, **typename** element,
typename ordering, **typename** policy>
void *siftdown*(position A, index i, element x, ordering *less*, policy& p);
Requirement: $A[i]$ contains a hole.

Effect: Insert element x into the sorted sequence stored on the siftdown path starting from the node with index i.

For each implementation alternative, only these functions should be modified.

Heap functions. Following the earlier recipes, the heap functions can be implemented easily using the policy functions and utility functions. This includes the *sort_heap*() function even though the more efficient implementation is completely independent of them. It should be noted that the heap functions are memoryless. If needed a new policy object is constructed, so no extra runtime information is passed from one invocation to another.

2.4 Benchmarks

Overall strategy. We will not repeat the experiments for *make_heap*() done in [BKS00], and those for *sort_heap*() (in-place mergesort) done in [SLK03]. Therefore, our focus is on the functions *push_heap*() and *pop_heap*(). To compare the efficiency of the implementation alternatives presented, we decided to use artificial operation generation in our experiments. Also, we restrict the problem sizes so that the heaps can be in internal memory all the time.

In earlier experimental studies on the efficiency of priority queues (see [Jon86, LL96, San00] and the references therein), several different operation-generation models were used. To describe the models we found interesting, we use I as a short-hand notation for *push_heap*(), D for *pop_heap*(), and M_n for *make_heap*() involving n elements. For nonnegative integers k and n, the models considered by us were:

Insert I^n: Determine the average running time of a single *push_heap*() when n operations are executed in all.

Hold $I^n(DI)^k$: Determine the average running time for a single iteration (DI) after inserting n elements.

Sort M_nD^n: Determine the average running time per element when sorting a sequence of n elements.

Peak $(IDI)^n(DID)^n$: Determine the average running time per operation.

Furthermore, the models involving element insertions should define how the new elements are generated.

Due to space restrictions we concentrate here on the sort model which we found most relevant. Directly, the sort model exercises only the *pop_heap*() function, but since many of the implementation alternatives are based on the bottom-up heapifying, the *push_heap*() function gets indirectly tested as well. We plan to report the performance of our programs in all the four models in a later version of this paper.

As explained in the introduction, we decided to use four kinds of input: unsigned ints; bigints [BM00, chap. 12]; unsigned ints with an expensive ordering function; and pairs containing unsigned ints as keys and bigints as an extra load, comparing the keys with an expensive ordering function. Uniformly generated random unsigned ints (32 bits) and bigints (strings of about 10 digits) were used all over. To give an idea of the execution times of various primitive operations, we accessed a vector containing unsigned ints or bigints both sequentially (stride $p = 1$) and arbitrarily (stride $p = 617$). The results of these micro benchmarks are shown in Table 2.3.

Test environment. To understand the development in computer hardware, we used three computers for our benchmarking having an Intel Pentium II (300 MHz), Intel Pentium III (1 GHz), and Intel Pentium 4 (1.5 GHz) processor, respectively. The experiments, the results of which are reported in this paper, were carried out on the latest computer (1st-level cache: 8 KB, 8-way associative 2nd-level cache: 256 KB, internal memory: 256 MB) running under Red Hat Linux 7.1 and using `g++` C++ compiler 3.0.4 with option `-O6`.

The experiments were carried out using the benchmark tool developed for the CPH STL by Katajainen and others [Dep06]. This tool is written in Python; most other scripting was also done in Python. The tool was used to generate test drivers which measure the CPU time spent by given operation sequences. Each experiment was repeated several times depending on the clock precision and the median of the execution times was reported if the execution times of 90% of the runs were within 20% of the reported value; otherwise, the whole experiment was ignored. Based on the scripts created it should be relatively easy for others to repeat our experiments in other environments.

Results. In order to understand the conflicting results reported by LaMarca and Ladner [LL96] and Sanders [San00], we analysed the test arrangements and found the following differences:

	LaMarca and Ladner [LL96]	Sanders [San00]
approach	top-down	bottom-up
element type	64-bit integers	32-bit interger keys, 32-bit satellite data
operation generation	hold model	peak model
element generation	earlier minimum plus a random increment	random over the whole range
computer	Pentium and others	Pentium II and others

Table 2.3: Cost of some instructions on an Intel Pentium 4 workstation for **a)** unsigned ints and **b)** bigints; n denotes the size of the sequence accessed and p the stride used.

a)

initializations	*instruction*	*unsigned int*
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$a[i] \leftarrow x$	$n = 2^{10} .. 2^{24}$ 4.1–4.7 ns
$p \leftarrow 617$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$a[i] \leftarrow x$	$n = 2^{10} .. 2^{14}$ 7.3–8.9 ns $n = 2^{15}$ 12 ns $n = 2^{16}$ 29 ns $n = 2^{16} .. 2^{22}$ 62–63 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$x \leftarrow a[i]$	$n = 2^{10} .. 2^{24}$ 3.3–3.8 ns
$p \leftarrow 617$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$x \leftarrow a[i]$	$n = 2^{10} .. 2^{15}$ 3.3–4.1 ns $n = 2^{16}$ 23 ns $n = 2^{17} .. 2^{22}$ 45–55 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$r \leftarrow (a[i] < x)$	$n = 2^{10} .. 2^{24}$ 5.3–5.8 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$r \leftarrow (\ln(a[i]) < \ln(x))$	$n = 2^{10} .. 2^{24}$ 580–610ns

b)

initializations	*instruction*	*bigint*
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$a[i] \leftarrow x$	$n = 2^{10} .. 2^{21}$ 60–66 ns $n = 2^{22}$ 290 ns
$p \leftarrow 617$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$a[i] \leftarrow x$	$n = 2^{10} .. 2^{12}$ 75–78 ns $n = 2^{13}$ 117 ns $n = 2^{14}$ 229 ns $n = 2^{15} .. 2^{20}$ 297–318 ns $n = 2^{21} .. 2^{22}$ 748–752 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$x \leftarrow a[i]$	$n = 2^{10} .. 2^{22}$ 18–21 ns
$p \leftarrow 617$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$x \leftarrow a[i]$	$n = 2^{10} .. 2^{12}$ 24 ns $n = 2^{13}$ 83 ns $n = 2^{14}$ 180 ns $n = 2^{15} .. 2^{22}$ 230–260 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$r \leftarrow (a[i] < x)$	$n = 2^{10} .. 2^{22}$ 13–16 ns

The most important differences are in the operation and element generation. In the experiments by LaMarca and Ladner the final destination of the new elements tends to be close to a leaf, whereas in those by Sanders the new elements tend to traverse higher up the heap.

To repeat Sanders' experiments, we downloaded the programs from his homepage and tested them on our three Pentium computers. On an Intel Pentium II computer for the peak model our results were similar to his, but for the sort model on all computers and for the peak model on the other computers the results were similar to those reported by LaMarca and Ladner.

Our initial profiling showed that the policy function *top_all_present*() is a bottleneck in the *pop_heap*() function. Therefore, we tried to implement it carefully for all implementation alternatives so that branch prediction would be easy. In general, we avoided **if-else**-constructs since these may make branch prediction harder. In this point we also observed that sometimes it was faster to use the conditional operator **?:** instead of a normal **if**-statement. After inspecting the assembler code, it turned out that our compiler could inline the function if the conditional operator was used, but with a normal **if**-statement it did not do that. In spite of this observation, we decided to keep the **if**-statements in our programs and leave the inlining for later code tuning.

In our preliminary experiments we tested all the implementation alternatives mentioned for the *pop_heap*() function. For integer data the best top-down approach was equally fast as the basic bottom-up approach, but when element comparisons are expensive the top-down approaches behaved badly. Therefore, we did not consider the top-down approaches any further.

We carried out the final experiments in two rounds. In the first round, for each implementation alternative and for each kind of input we determined the best arity. If an alternative was the fastest for more than 80% of the data points measured, we declared it as a ***winner***. In the second round, for each kind of input the winners were compared against each others. The results of these tests are shown for the four types of input in Figs. 3.3, 3.4, 3.5, and 2.5, respectively. In all experiments we used the functions *partial_sort*() (carefully coded heapsort using binary heaps and bottom-up heapifying) and *sort*() (introsort) from the Silicon Graphics Inc. implementation of the STL [Sil04] as our reference implementations.

Discussion. For integer data our results are similar to those reported by LaMarca and Ladner [LL96, LL99] even though on our Pentium 4 computer the point when the 4-ary heaps become faster than 2-ary heaps appears first at the end of the range plotted in Figure 3.3. In all fairness other data types have to be considered, too. Our results show that a single arity is not the best for all kinds of elements and all kinds of ordering functions. Furthermore, no heapifying approach is the best for all kinds of elements and all kinds of ordering functions, but for randomly generated inputs it is very difficult to beat the basic bottom-up approach.

Of the approaches improving the worst-case performance of the basic bottom-up heapifying, the two-levels-at-a-time bottom-up heapifying is simple and performs better than those based on binary or exponential binary search. Moreover,

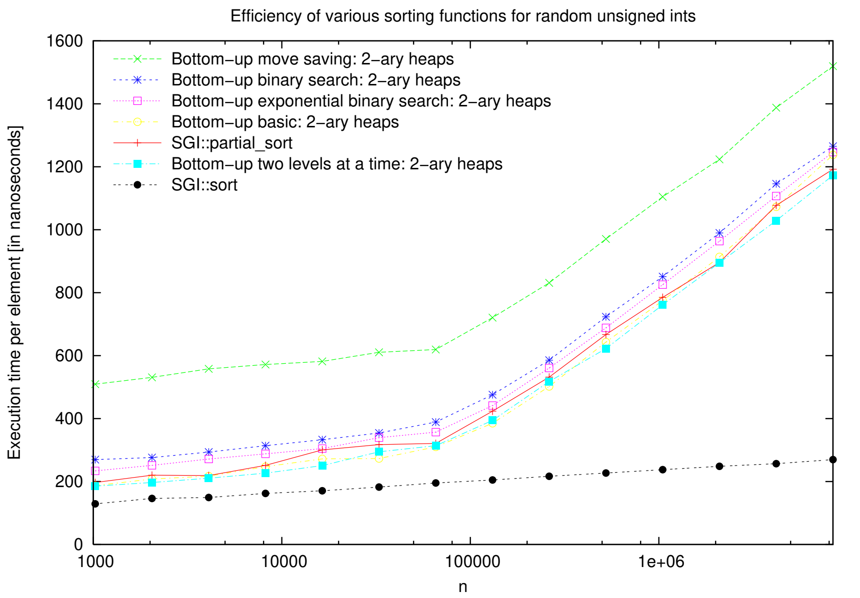

Figure 2.2: Performance of the first-round winners for random unsigned ints on an Intel Pentium 4 workstation.

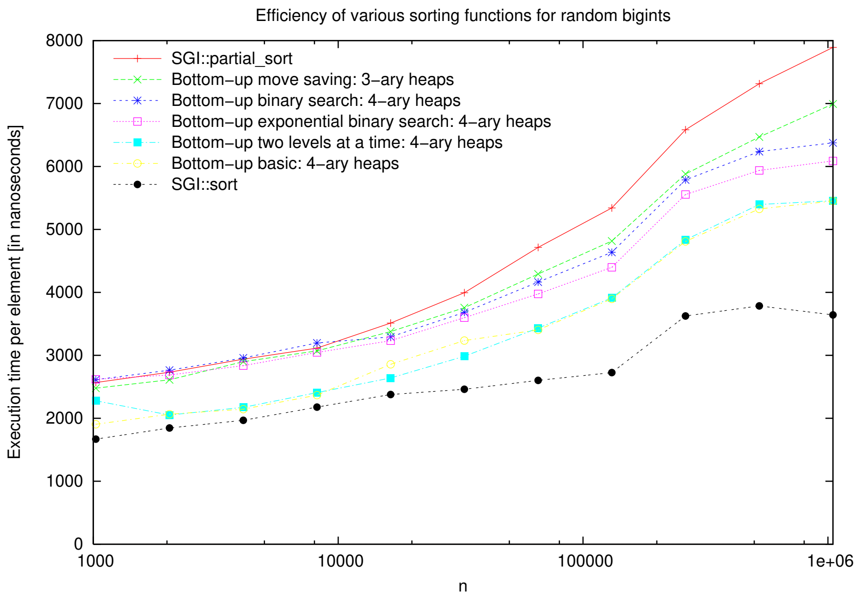

Figure 2.3: Performance of the first-round winners for random bigints on an Intel Pentium 4 workstation.

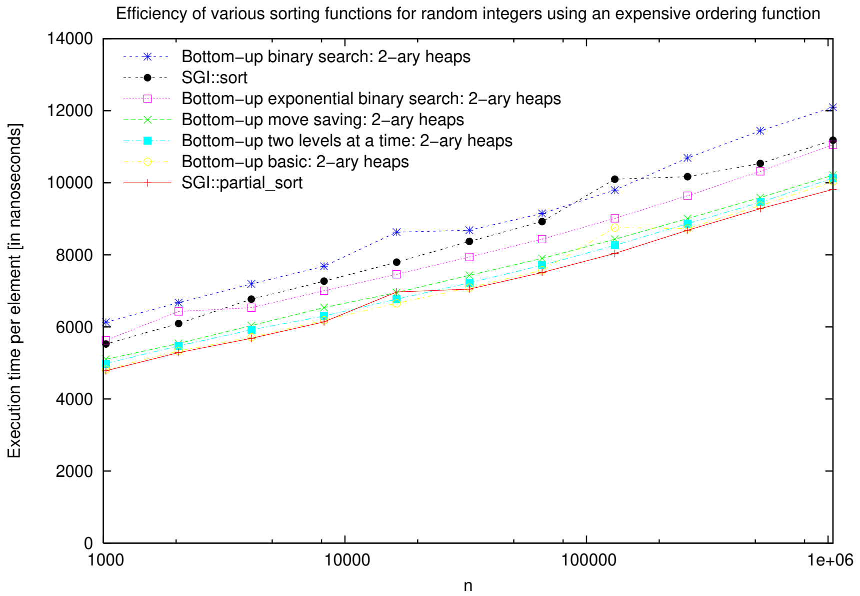

Figure 2.4: Performance of the first-round winners for random unsigned ints with an expensive ordering function on an Intel Pentium 4 workstation.

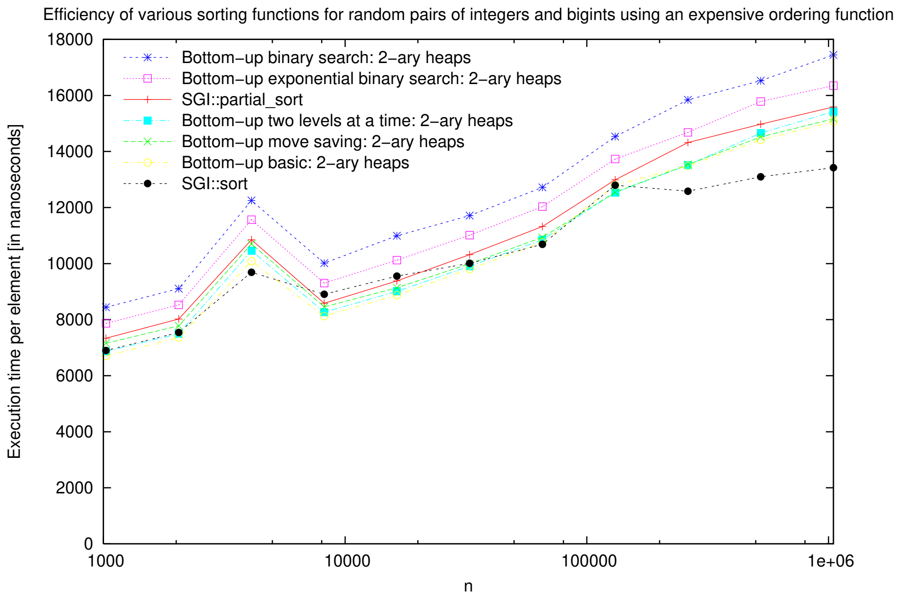

Figure 2.5: Performance of the first-round winners for random pairs of unsigned ints and bigints with an expensive ordering function on an Intel Pentium 4 workstation.

for small problem sizes (say for $n \leq 2^{20}$) the difference in the worst-case number of element comparisons is small for these three approaches, so we recommend that the two-levels-at-a-time bottom-up heapifying is used as a basis for a tuned implementation. Doubling in the worst-case number of element moves may be significant, but according to our experiments move saving seems not to be worthwhile because typically only a few levels are traversed up in each *push_heap*() and *pop_heap*() operation. If it is important to perform fewer element moves, one may consider using extra space (pointers) when implementing the priority queue class since there this is allowed.

Software availability

All source code and scripts developed in the course of this study are available at the CVS repository for the CPH STL project [Dep06].

Acknowledgements

We thank Tomi A. Pasanen for proposing the use of exponential binary search to improve the worst case of the bottom-up heapifying.

References

[Ale01a] Andrei Alexandrescu. Generic<programming>: A policy-based `basic_string` implementation. *C++ Expert Forum*, 2001. Available at `http://www.cuj.com/experts/1096/toc.htm`.

[Ale01b] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Pattern Applied*. Addison-Wesley, Upper Saddle River, 2001.

[BKS00] Jesper Bojesen, Jyrki Katajainen, and Maz Spork. Performance engineering case study: heap construction. *The ACM Journal of Experimental Algorithmics*, 5:Article 15, 2000.

[BM00] Dov Bulka and David Mayhew. *Efficient C++: Performance Programming Techniques*. Addison-Wesley, Reading, 2000.

[Boj98] Jesper Bojesen. Heap implementations and variations. Written project, Department of Computing, University of Copenhagen, Copenhagen, 1998. Available at `http://www.diku.dk/forskning/performance-engineering/Perfeng/resources.html`.

[Bri03] *The C++ Standard: Incorporating Technical Corrigendum 1*. John Wiley and Sons, Ltd., 2nd edition, 2003.

[BY76] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5:82–87, 1976.

[Car87] Svante Carlsson. A variant of Heapsort with almost optimal number of comparisons. *Information Processing Letters*, 24:247–250, 1987.

[Car92] Svante Carlsson. A note on Heapsort. *The Computer Journal*, 35:410–411, 1992.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, 2nd edition, 2001.

[Dep06] Department of Computing, University of Copenhagen. The CPH STL. Website accessible at `http://www.cphstl.dk/`, 2000–2006.

[ES02] Stefan Edelkamp and Patrick Stiegeler. Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ comparisons. *The ACM Journal of Experimental Algorithmics*, 7:Article 5, 2002.

[Flo64] Robert W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7:701, 1964.

[GM86] Gaston H. Gonnet and J. Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15:964–971, 1986.

[GZ96] Xunrang Gu and Yuzhang Zhu. Optimal heapsort algorithm. *Theoretical Computer Science*, 163:239–243, 1996.

[Jen01] Brian S. Jensen. Priority queue and heap functions. CPH STL Report 2001-3, Department of Computing, University of Copenhagen, Copenhagen, 2001. Available at `http://www.cphstl.dk/`.

[Joh75] Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4:53–57, 1975.

[Jon86] Douglas W. Jones. An emprirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:300–311, 1986.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley Longman, Reading, 2nd edition, 1998.

[KPT96] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3:27–40, 1996.

[KV03] Jyrki Katajainen and Fabio Vitale. Navigation piles with applications to sorting, priority queues, and priority deques. *Nordic Journal of Computing*, 10:238–262, 2003.

[LL96] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics*, 1:Article 4, 1996.

[LL99] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31:66–104, 1999.

[Mus97] David R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27:983–993, 1997.

[NBC+95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4:603–627, 1995.

[Oko80] Seiichi Okoma. Generalized Heapsort. In *Proceedings of the 9th Symposium on Mathematical Foundations of Computer Science*, volume 88 of *Lecture Notes in Computer Science*, pages 439–451, Berlin/Heidelberg, 1980. Springer-Verlag.

[Pas] Tomi A. Pasanen. Public communication, December 2003.

[San00] Peter Sanders. Fast priority queues for cached memory. *The ACM Journal of Experimental Algorithmics*, 5:Article 7, 2000.

[Sil04] Silicon Graphics, Inc. Standard template library programmer's guide. Website accessible at `http://www.sgi.com/tech/stl/`, 1993–2004.

[SLK03] Jakob Sloth, Morten Lemvigh, and Mads Kristensen. Sortering i CPH STL. CPH STL Report 2003-2, Department of Computing, University of Copenhagen, Copenhagen, 2003. Available at `http://www.cphstl.dk/`.

[Weg93] Ingo Wegener. Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). *Theoretical Computer Science*, 118:81–98, 1993.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[WT89] L. M. Wegner and J. I. Teuhola. The external heapsort. *IEEE Transactions on Software Engineering*, 15:917–925, 1989.

Appendix

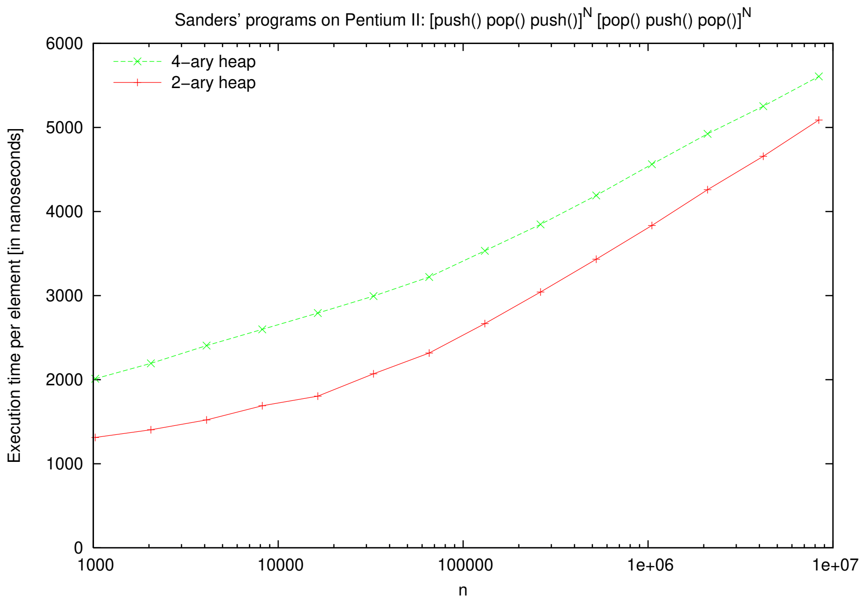

Figure 2.6: Performance of the Sanders heap programs for random unsigned ints on an Intel Pentium 2 workstation using the peak model.

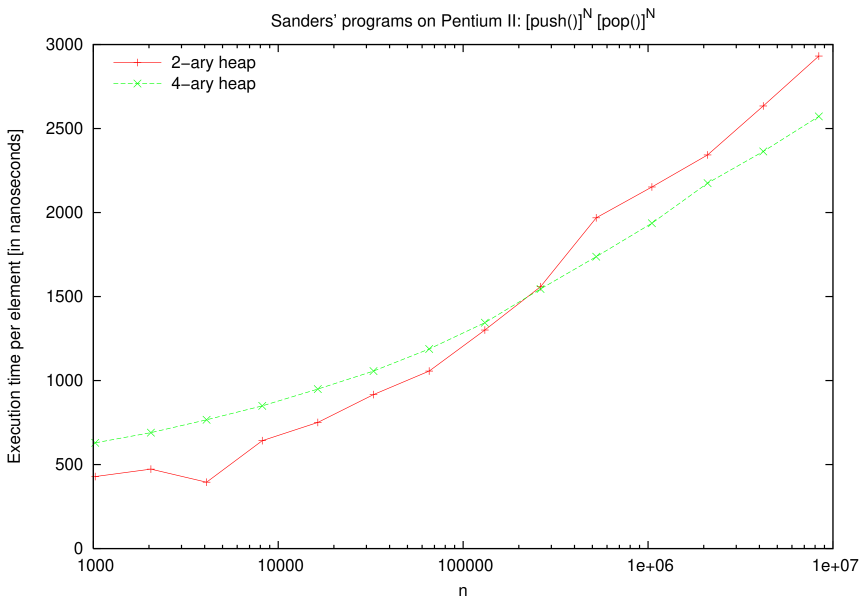

Figure 2.7: Performance of the Sanders heap programs for random unsigned ints on an Intel Pentium 2 workstation using the sort model.

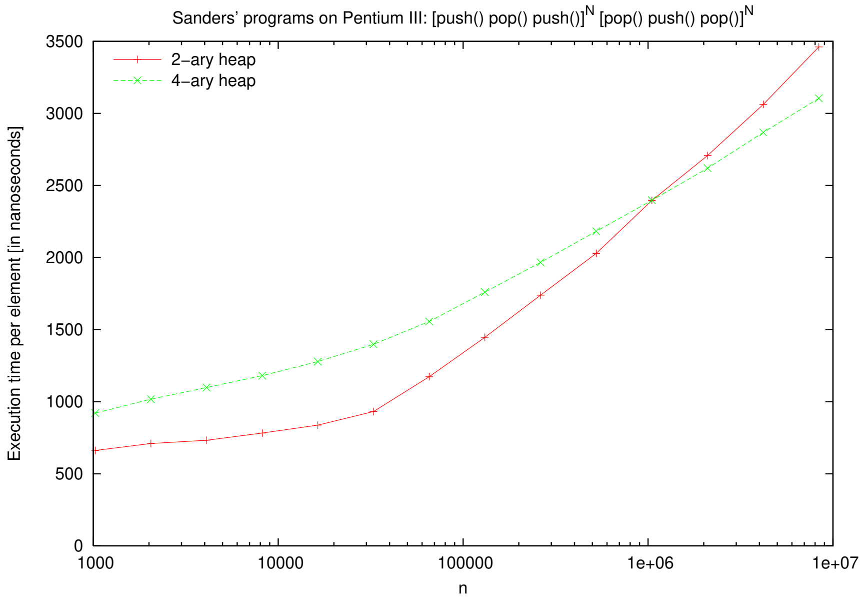

Figure 2.8: Performance of the Sanders heap programs for random unsigned ints on an Intel Pentium 3 workstation using the peak model.

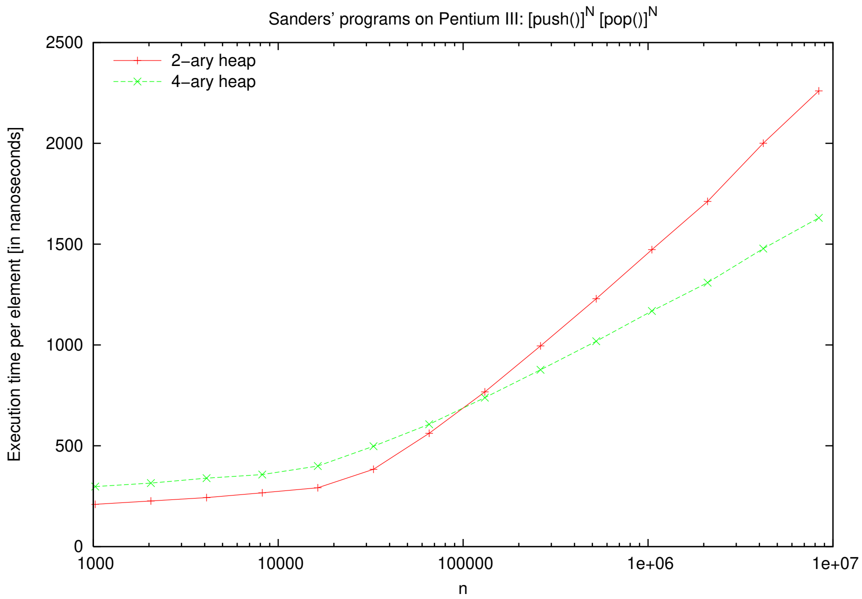

Figure 2.9: Performance of the Sanders heap programs for random unsigned ints on an Intel Pentium 3 workstation using the sort model.

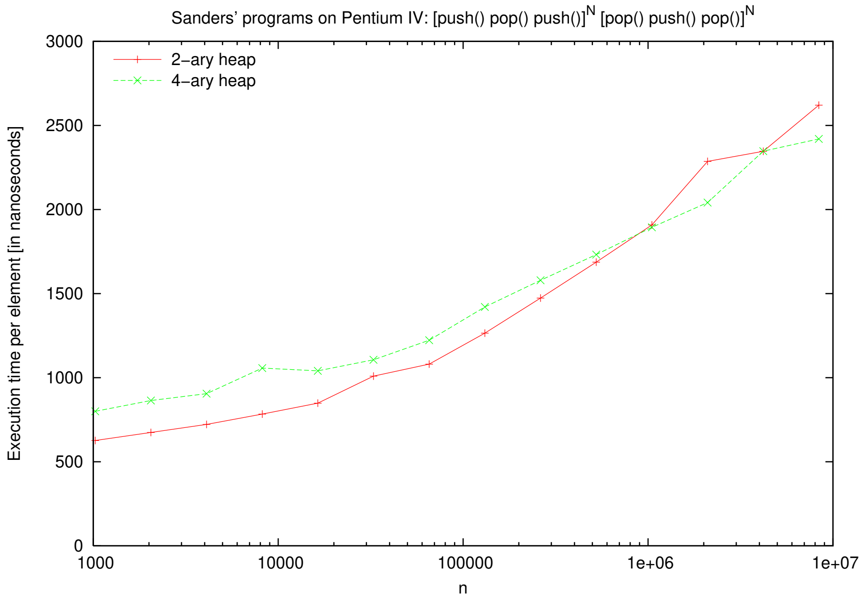

Figure 2.10: Performance of the Sanders heap programs for random unsigned ints on an Intel Pentium 4 workstation using the peak model.

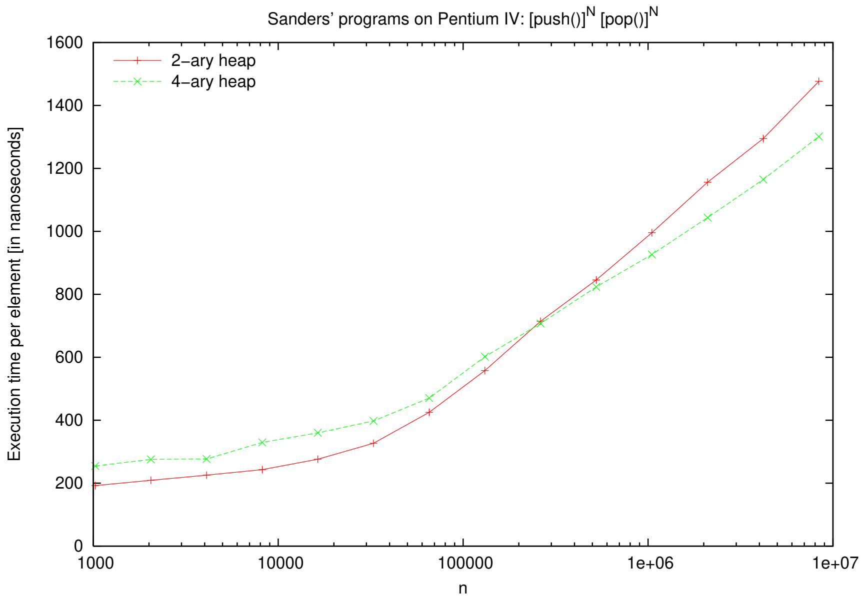

Figure 2.11: Performance of the Sanders heap programs for random unsigned ints on an Intel Pentium 4 workstation using the sort model.

Chapter 3

An experimental evaluation of navigation piles

Abstract. A navigation pile, which can be used as a priority queue, is an extension of a selection tree. In a compact form the whole data structure requires only a linear number of bits in addition to the elements stored. In this paper, we study the practical efficiency of three different implementations of navigation piles and compare their efficiency against two implementations of binary heaps. The results of our experiments show that navigation piles are a good alternative to heaps when element moves are expensive—even if heaps store pointers to elements instead of elements. Based on the experimental comparison of the three navigation-pile implementations it is clear that care should be taken when applying space saving strategies that increase the number of instructions performed. In addition to our experimental findings, we give a new and simple way of dynamizing a static navigation pile. Furthermore, we introduce a pointer-based navigation pile which is inherently dynamic in its nature and can be made to support deletions as well.

3.1 Introduction

A *priority queue* is a data structure which stores a collection of elements and supports the operations *construct*, *push*, *pop*, and *top* in terminology used in the C++ standard [Bri03, §23.2]. A maximum element of the priority queue is selected with respect to an ordering function given at the time of construction. For the realization of a priority queue, the navigation pile, introduced by Katajainen and Vitale [KV03], is an alternative to the standard binary heap [Wil64] and similar data structures. In its basic form, a navigation pile is a static data structure where the maximum number of elements to be stored must be known in advance. The main advantage of navigation piles is that they provide fast worst-case priority-queue operations and have low space requirements (linear number of bits in addition to the elements stored). The same holds true even if the data structure is made fully dynamic.

At the beginning of this study we wanted to obtain a greater understanding of the practical utility of compact navigation piles and to study the practical utility of other adaptations of that data structure. In this paper we report the results of our experiments where we consider three different implementations of navigation piles. The first implementation, called a *compact pile*, is based on the original description given in [KV03]. The elements are stored in a resizable array and above this array a bit container is built which stores navigation information in packed form. In the second implementation the bit container is replaced with an index array so we call it an *index pile*. In the third implementation the whole data structure is realized using concrete nodes and pointers; from now on we call it a *pointer-based pile*.

When analysing the runtime complexity of priority-queue operations and the space complexity of data structures, we use *word RAM* as our model of a computer (for a precise definition of the model, see [Hag98]). In an *element comparison* the relative order of two elements is determined by evaluating the specified ordering function, which defines a strict weak ordering on the set of elements manipulated (for a definition of strict weak ordering, see, for example, [Bri03, §25.3]). By an *element move* we mean the execution of a copy operation, copy construction or copy assignment, invoked for copying elements. We use the term *cost* to denote the sum of word-RAM instructions, element constructions, element destructions, and element comparisons performed.

In the three implementations studied, there is an interesting tradeoff between the amount of space and instructions used. Let N denote the capacity of a navigation pile and n the number of elements stored. A pointer-based pile uses $5n + O(\log_2 n)$ words to store all navigation information and simple pointer operations are performed when moving around in the data structure. As a sharp contrast, a compact pile uses $\lceil 2N/w \rceil$ words of space, where w denotes the length of each machine word measured in bits, but it requires more complicated calculations in navigation through the pile structure. As a compromise, an index pile uses $N - 1$ words to store the navigation information and does less calculations than a compact pile.

The specific questions addressed by this study are the following: Can navigation piles be considered an alternative to the standard heap when implementing priority queues? Can a data structure based on pointers be competitive in a contemporary computer where the locality of data is important due to a hierarchical memory structure? Does the extra computational work, in form of extra instructions used in connection with packing and unpacking information, counterbalance the advantage gained by reducing the use of extra space?

Besides the experimental results, the following theoretical contributions presented in this paper are new: 1) the introduction of pointer-based piles, 2) the introduction and use of the first-ancestor technique for the realization of *pop*, and 3) the introduction of a new way of dynamizing a static navigation pile.

Navigation piles in a nutshell. In order to define a navigation pile, which is a binary tree, we use the following standard terminology for trees. A node of a

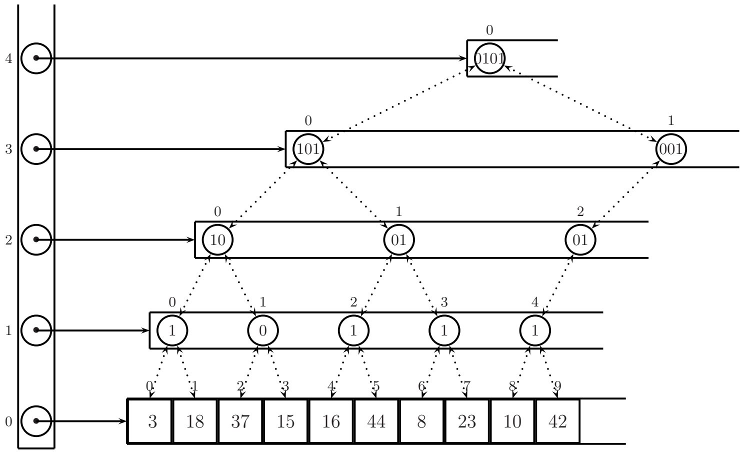

Figure 3.1: A dynamic compact pile storing 10 integers. For example, the bits $(001)_2$ in the second branch of height three indicate that the largest integer in the leaves of the subtree rooted at that branch is 42; the leaf has position 1 among the leaves spanned; indexing starts from 0. The offset of the spanned subarray from the beginning of the array is calculated from the height (h) and index (i) of the branch using the formula $i \times 2^h$.

tree is the *root* if it has no parent, a *leaf* if it has no children, and a *branch* if it has at least one child. The *depth* of a node is the length of the path from that node to the root. The *height* of a node is the length of the longest path starting from that node and ending at a leaf. In a *complete binary tree* all branches have two children and all leaves have the same depth.

A navigation pile can be seen as an extension of a selection tree described, for example, in [Knu98]. In a *navigation pile* elements are stored at the leaves, and each branch stores a reference to the maximum element held in the leaves of the subtree rooted at that branch. The references at branches, referred to as *navigation information*, can be stored in different ways. According to the original proposal of Katajainen and Vitale [KV03], navigation information is stored in a packed form, but each branch could also store an index of the corresponding maximum element in the resizable array storing the elements.

Our implementation of a compact pile follows closely the guidelines given in [KV03]. An index pile is a natural extension where the packing/unpacking of references is omitted. Both data structures are static so the maximum capacity N must be known beforehand. A dynamic navigation pile, as described in [KV03], is a collection of static navigation piles. Therefore, the performance of our static implementations can be used as a baseline for the performance of the dynamic version as well.

The data-structural transformation described in [KV03] uses at most logarithmic number of static data structures to represent a dynamic data structure. This approach is theoretically sound, but tedious to implement. Therefore, we describe a new, simpler, and more direct way of dynamizing the data structure. The new strategy is to use one resizable array to store the branches of the same height and maintain a header which stores references to these resizable arrays (for an illustration, see Figure 3.1). That is, there can be at most $\lceil \log_2 n \rceil$ resizable arrays and the objects stored at each array are of the same size even if packing is used. The header itself is also a resizable array. A similar dynamization strategy has previously been used in connection with deques (see [KM01]). Even though this strategy requires some administrative work for maintaining the header and allocating/deallocating arrays "on the fly", the extra work should not increase the cost of the priority-queue operations significantly. The complexity of the operations used to navigate through a navigation pile, as for instance the calculation of the first and second child of a branch, is not increased and, therefore, these operations should not increase the cost of the priority-queue operations.

If N denotes the capacity of a static data structure and n the number of elements stored prior to each priority-queue operation, a compact pile and an index pile give the following performance guarantees [KV03]:

- *construct* requires $n-1$ element comparisons, n element moves, and $O(n)$ instructions.
- *top* requires $O(1)$ instructions.
- *push* requires $\log_2 \log_2 n + O(1)$ element comparisons, one element move, and $O(\log_2 n)$ instructions.
- *pop* requires $\lceil \log_2 n \rceil$ element comparisons, two element moves, one element destruction, and $O(\log_2 n)$ instructions.

Excluding the space required for storing the elements, a compact pile requires at most $2N$ bits of additional space to store the navigation information. If the $2N$ extra bits are packed, the navigation information uses $\lceil 2N/w \rceil$ words in total, w denoting the size of a machine word measured in bits. An index pile requires at most $N-1$ words of additional space, one index per branch.

A pointer-based pile, where nodes are stored explicitly and all connections are handled by pointers, is automatically a dynamic data structure giving the following performance guarantees:

- *construct* requires $n-1$ element comparisons, n element moves, and $O(n)$ instructions.
- *top* requires $O(1)$ instructions.
- *push* requires $\log_2 \log_2 n + O(1)$ element comparisons, one element move, and $O(\log_2 n)$ instructions.
- *pop* requires $\lceil \log_2 n \rceil$ element comparisons, one element destruction, and $O(\log_2 n)$ instructions.

In addition to the space required for storing the elements, a pointer-based pile requires at most $5n + O(\log_2 n)$ words of extra space to store the pointers, which could be reduced to $4n + O(\log_2 n)$ words by using the child-sibling representation of binary trees (see, e.g. [Tar83, Section 4.1]).

Experimental setting. As an immediate contender of our implementations of navigation piles, we chose the priority-queue implementation relying on implicit binary heaps implementation of the C++ standard library (shipped with the g++ compiler version 3.3.4 which is available at the Free Software Foundation, Inc.). This heap implementation is known to be highly tuned; from now on we refer to this as an *implicit heap*. Implicit binary heaps have also been used in many of the earlier experimental studies (see, for example, [Jon86, LL96, LL99, San00]), which make it possible to compare our results indirectly to these earlier results.

One problem with an implicit heap is that it does not provide *referential integrity*, i.e. it does not keep external references to elements inside the data structure valid, which is important if the structure is to be extended to support general erasure (*erase*) or modification (e.g. *decrease*) of elements. As an opposite, a pointer-based pile naturally provides referential integrity and can easily be extended to support *erase*. To make the comparison between pointer-based piles and heaps more fair, we extended the C++ library heap with the ability to support referential integrity; this implementation is from now on referred to as a *referent heap*. To support referential integrity, an adapter class was constructed. This adapter class operates with pointers instead of elements and the elements have references back to the pointers in the heap. In this way only pointers are moved, not elements, and external references to elements remain valid.

A generic priority queue, as defined in the C++ standard [Bri03], should be able to perform well under various circumstances. It should handle built-in types, user-defined types, and different types of ordering functions efficiently. In an attempt to cover the effects of a broad selection of possible input parameters with a reasonable number of experiments, we chose input parameters that represent a variation of cheap and expensive element comparisons and cheap and expensive element moves.

In the experiments the following types of input parameters were used: 1) built-in unsigned integers; 2) bigints, as described in the book of Bulka and Mayhew [BM00], which represent an unsigned integer as a string of digits; and 3) built-in unsigned integers combined with an ordering function that computes the natural logarithm of the elements before comparing them. In the case of built-in unsigned integers, both element comparisons and element moves are cheap. In the case of bigints, element comparisons are cheap, but the element moves are expensive. In the last case element comparisons are expensive, but the element moves are cheap. In earlier studies, similar settings for input parameters have been used. LaMarca and Landner [LL96, LL99] used 32-bit and 64-bit integers and Sanders [San00] used 32-bit integer keys with a 32-bit satellite data attached to the key. In experimental studies on sorting, elements consisting of 100-byte records with 10-byte keys are often used (see, e.g. [NBC$^+$95]). Edelkamp and Stiegeler [ES02] used a number of expensive ordering functions, of which the expensive ordering function used by us is one.

To make the experiments reflect different scenarios of use, we employed several different models for the generation of priority-queue operations:

Insert: Measure the average running time per operation in a sequence of n *push* operations.

Peak: Measure the average running time per operation in a sequence of n (*push pop push*) operations followed by a sequence of n (*pop push pop*) operations.

Sort: Measure the average running time per element for a single *construct* operation for n elements followed by n *pop* operations.

Hold: Measure the average running time per operation in a sequence of k (*push pop*) operations after a sequence of n *push* operations has already be done. In this model only input data of type float are considered.

These operation-generation models have also been used in several other experimental studies of priority queues (see [Jon86, LL96, San00] and the references therein).

3.2 Our implementations of navigation piles

In this section, we give a general description of navigation piles and all the three implementations considered in this study. The nodes of a navigation pile are divided into two groups: leaves which hold the elements and branches which hold the navigation information used when locating the maximum element stored in a subtree. The overall maximum element stored in the data structure can be found by following the reference at the root.

A navigation pile could be defined recursively as follows. Let n denote the number of elements being stored. If $n = 1$, the data structure has only one leaf which stores the single element and there are no branches. If $n = 2$, the elements are stored in their respective leaves and there is a single branch which contains a reference to the leaf storing the maximum of the two elements. Assume now that $n > 2$ and let h be the largest positive integer such that $2^h < n$. Two navigation piles S and T are constructed recursively; S contains the first 2^h elements and T contains the remaining $n - 2^h$ elements. If the height of T is less than that of S, T is transformed to a tree having the same height as S by incrementally creating a new parent for the current root and making the old root the left child of the new root, and this process is repeated until both trees have the same height. In each step the new root should contain the same navigation information as the former root. When the height of S and T is the same, a new branch r is created and this node becomes the root of the joined tree; S becomes the left subtree of r and T the right subtree of r; and the navigation information of r is determined by comparing the elements referred to by the roots of S and T and setting the reference at r to point to the leaf storing the maximum of these two elements.

Representation. In a pointer-based pile, each leaf contains a parent pointer and an element. Each branch contains a parent pointer which can be null, a left-child pointer, a right-child pointer which can also be null, and a pointer to a leaf. To save space, the child pointers of branches are allowed to point to either a leaf or to a branch. Navigation through the data structure is done by following the pointers contained in the nodes. It should also be emphasized that this implementation of a navigation pile is fully dynamic without any modifications.

In an index pile, the nodes are implicit and there is an implicit interconnection between the indices of leaves and the indices of the resizable array storing the elements. The indices held in branches are stored in a separate container. Movement inside the data structure involves arithmetic operations on array indices. To simplify the program logic, a collection of utility functions was introduced, each having some specific function, e.g. *left-child* is an example of such a utility function.

In a compact pile, the nodes are also implicit, and there is the same interconnection between the leaves and elements stored in a resizable array as in an index pile. The navigation information held in branches is stored in a resizable bitarray. To compress the indices stored at branches as much as possible, for each branch we store a relative index of a leaf inside the leaf-subarray spanned by that branch. Using this relative index and the offset of the spanned subarray from the beginning of the element array, the actual position of the element referred to can be calculated (cf. Figure 3.1). For example, if a branch spans a group of four elements, the relative index can be stored as a number between zero and three which can be represented as a bit-pattern of size two. As a consequence of relative indexing the total amount of space needed for storing the navigation information can be brought down to $\lceil 2N/w \rceil$ words, N being the total capacity and w the size of a machine word. The utility functions for a compact pile are in many ways the same as those for an index pile; the main difference is the packing and unpacking of navigation information and the additional offset calculations required.

Priority-queue operations. To make the construction of a navigation pile more cache-friendly, *construct* is performed by visiting the branches in depth-first order. The actual implementation is iterative, instead of recursive, and nodes are visited in a bottom-up manner. The navigation information of a selected branch is computed by using the navigation information stored at the children of that branch and comparing the elements pointed to by the children. A special case in the construction is the computation of the navigation information for the branches having height one. The navigation information of these branches is computed through a comparison of the elements contained in the two leaves associated with the branch.

A new element is inserted by *push* into the first empty leaf. After this, the navigation information is updated (if neccessary) on the path from the corresponding leaf to the root, and new branches are initialized/created when necessary.

The maximum element to be returned by *top* is found by following the

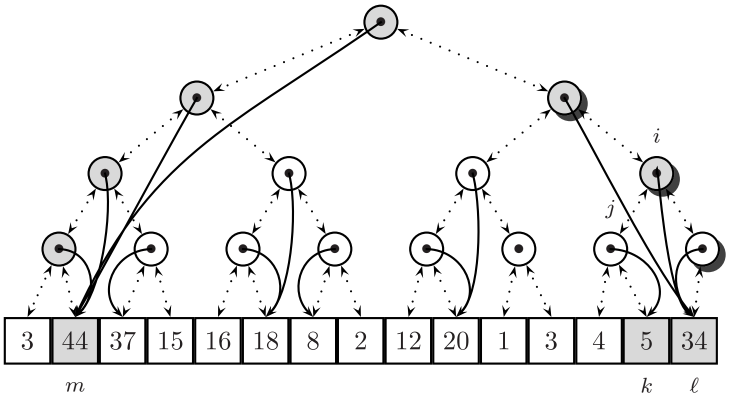

Figure 3.2: Illustration of the first-ancestor technique. The nodes, the contents of which may change, are indicated with light gray. When updating the contents of the shadowed branches on the right, no element comparisons are necessary.

reference stored at the root.

When *pop* is executed, the maximum element referred to by the root is erased and the navigation information is updated accordingly. This update is done in three different ways depending on the circumstances inside the navigation pile (for an illustration of one of the cases, see Figure 3.2). The implementation details of the update depend on the form of the navigation pile. For a compact pile and an index pile, *pop* is accomplished in a similar manner as follows.

Let ℓ be the last leaf and m the leaf containing the maximum element. Let i be the first ancestor of the last leaf which has two children. If the left child of i is a branch, let j be this node. Let k be the leaf referred to by j, or if j is undefined, let k be the left child of i.

Using the *first-ancestor technique*, *pop* is executed as follows. *Case* 1: $m = \ell$. The leaf containing the maximum element (the last leaf) is erased and the references at the branches on the path from the new last leaf to the root are updated by performing repeated element comparisons. The traversal up stops when a reference different from ℓ is met. *Case* 2: $m \neq \ell$ and i refers to k. The element stored at leaf ℓ is moved to leaf m, and the last leaf is erased. The references at the branches on the path from leaf m to the root are updated by performing repeated element comparisons. *Case* 3: $m \neq \ell$ and i refers to ℓ (see Figure 3.2). The element in leaf k is moved to leaf m, the element in leaf ℓ is moved to leaf k, and the last leaf is erased. The branches on the path from i to the root are assigned to refer to leaf k. The references at the branches on the path from leaf m to the root are updated by performing repeated element comparisons.

Let us now consider *pop* for a pointer-based pile. The main difference is that now whole nodes are moved instead of elements. As before, there are three cases. *Case* 1: $m = \ell$. The leaf containing the maximum element (the last leaf) is erased and the navigation information in the branches on the path from the new last leaf to the root is updated. *Case* 2: $m \neq \ell$ and i refers to k. The last leaf ℓ is moved into the place of leaf m, and leaf m is erased. The navigation information at the branches on the path from leaf ℓ to the root is updated. *Case* 3: $m \neq \ell$ and i refers to ℓ. Leaf k is moved to the position of leaf m, leaf ℓ is moved to the earlier position of leaf k, and leaf m is erased. The navigation information of the branches on the path from leaf l up to branch i, but no including i, is assigned to refer to leaf ℓ. The navigation information of the branches on the path from leaf k to the root is updated by performing repeated element comparisons.

3.3 Experimental setup

In our experiments we compared the performance of the three versions of navigation piles in order to determine whether an implementation, where the navigation information is packed, gives a performance advantage over a non-packed implementation; and whether a pointer-based implementation have a performance advantage over an implicit implementation, or vice versa. Also, we compared the performance of our implementations against two heap implementations. The first implicit heap implementation was taken directly from the standard library available at our environment (that shipped with the g++ compiler). The other referent heap implementation is our modification which provides referential integrity. A referent heap is simply an adaptor which stores the elements and passe pointers to a standard heap.

In the experiments we used the three types of input parameters mentioned in the introduction: unsigned integers (32 bit), bigints (strings of about 10 digits), and unsigned integers combined with an expensive ordering function. Integers and bigints were generated randomly.

We ran our experiments for the four operation-generation models: insert, peak, sort, and hold (see the introduction for a description of these models), but as a consequence of the space restrictions we only include the benchmark results for the sort model in the main part of this paper, and present the results of the other models in the appendix.

We performed the experiments in the following environment:

Hardware: dual CPU: Intel Pentium 4 (3 GHz), cache: 1 MB, internal memory: 3.8 GB

Software: operating system: Gentoo, Linux kernel: 2.4.26, compiler: g++ 3.3.4, compiler option: -O6.

All time measurements were done using the benchmark tool (Benz) developed by Katajainen and extended by others (see [SP03]). Benz was configured to measure the CPU-time consumption of a given operation sequences. In the framework of Benz, each experiment is repeated several times until 90% of the

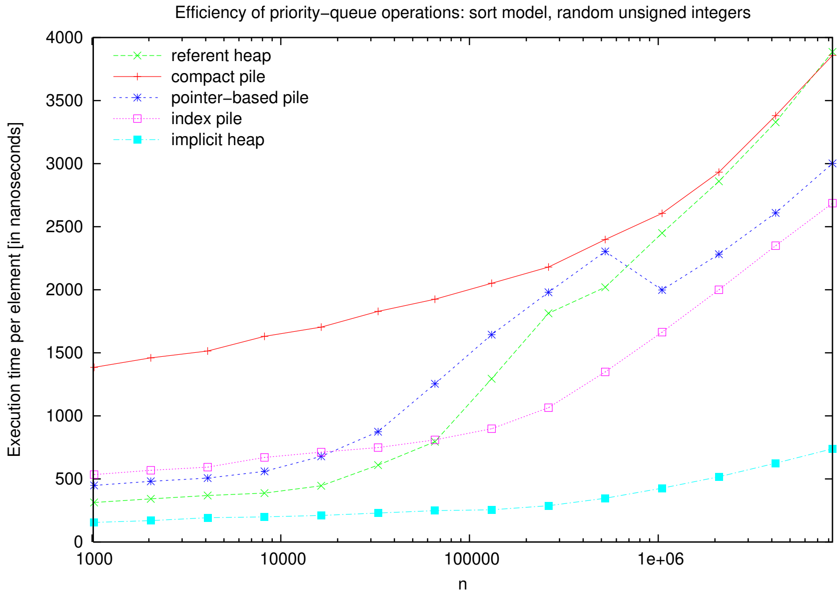

Figure 3.3: Performance of the implementations for the sort model using random unsigned integers on an Intel Pentium 4 workstation.

outcomes differ at most 20% from the median, which is reported; or more than 100 trials have been done after which the experiment is aborted.

3.4 Results

The results of our experiments for the sort model are given in Figures 3.3 (unsigned integers), 3.4 (bigints), and 3.5 (expensive comparisons). When manipulating integers the execution time for all variants of navigation piles is much higher than the execution time for implicit heaps, as shown by Figure 3.3. The execution time for referent heaps is lower than the execution time for pointer-based piles when the number of elements is small, but after the crossover point (at about 500 000 elements) the execution time for referent heaps is higher. If referential integrity should be provided, pointer-based piles are a viable alternative to heaps even for integer input data. The tendency seen in the results for integer data is repeated in the experiments where an expensive ordering function is applied, but for large input sizes the differences in execution times decrease among the implementations. As seen in Figure 3.4, the advantages of navigation piles compared to implicit heaps become apparent when the elements considered are large and element moves become expensive.

The extra space used by pointer-based piles in comparison with that used by index piles has only a small effect on the performance when the number of elements is small, but as the number of elements grows the execution time

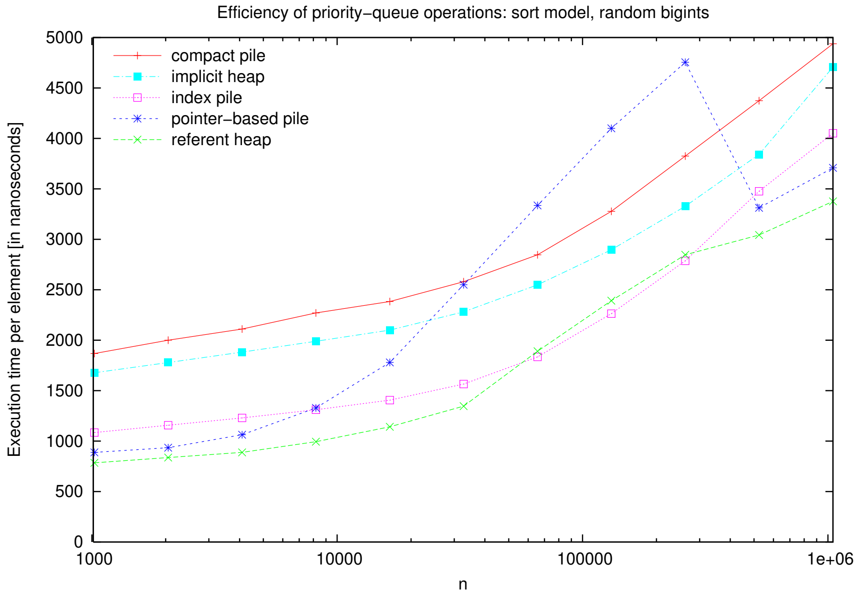

Figure 3.4: Performance of the implementations for the sort model using random bigints on an Intel Pentium 4 workstation.

for pointer-based piles increases faster than the execution time for index piles. After observing that the use of extra space has a negative consequence on the performance, it could be expected that saving space by packing the navigation information would give better performance, at least when the number of elements increases. However, this does not seem to be the case; our results show that the increase in the instruction count due to packing devaluates the performance advantage gained by saving space.

3.5 Guidelines

When studying the results of our experimental results for the three implementations of the navigation-pile data structure, it can be seen that they not only answer the specific questions stated, but they also provide some general guidelines for the use of extra space versus an increase in the instruction count. It seems, that saving space can have a high cost on the performance if this at the same time increases the number of instructions performed which is often the case for many space-saving strategies. On the other hand, as indicated by the implicit-heap implementation, if a space-saving strategy does not increase the instruction count significantly, it may be possible to get better performance.

Comparing the results obtained for navigation piles and for implicit heaps, it can be seen that, when the elements being manipulated are large and element moves thereby become expensive, it is appropriate to increase the code

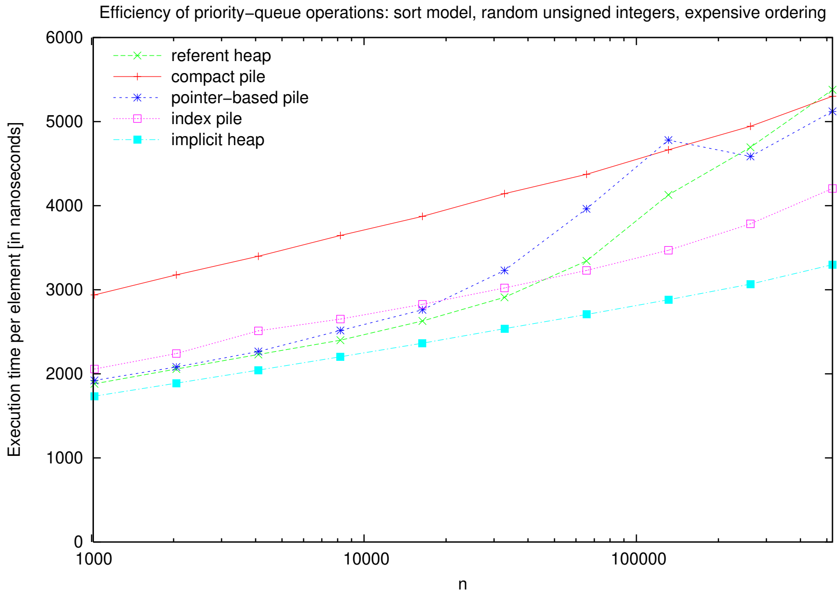

Figure 3.5: Performance of the implementations for the sort model using random unsigned integers with an expensive ordering function on an Intel Pentium 4 workstation.

complexity if one thereby can reduce the number of element moves performed.

If a priority queue has to support referential integrity, our experimental results show that other types of priority queues than an implicit heap should be taken into consideration.

Software availability

The programs used in this experimental study are accessible via the home page of the CPH STL project [Dep06].

References

[BM00] Dov Bulka and David Mayhew. *Efficient C++: Performance Programming Techniques*. Addison-Wesley, Reading, 2000.

[Bri03] *The C++ Standard: Incorporating Technical Corrigendum 1*. John Wiley and Sons, Ltd., 2nd edition, 2003.

[Dep06] Department of Computing, University of Copenhagen. The CPH STL. Website accessible at `http://www.cphstl.dk/`, 2000–2006.

[ES02] Stefan Edelkamp and Patrick Stiegeler. Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ comparisons. *The ACM Journal of Experimental Algorithmics*, 7:Article 5, 2002.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

[Jon86] Douglas W. Jones. An emprirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:300–311, 1986.

[KM01] Jyrki Katajainen and Bjarke Buur Mortensen. Experiences with the design and implementation of space-efficient deques. In *Proceedings of the 5th Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes in Computer Science*, pages 39–50, Berlin/Heidelberg, 2001. Springer-Verlag.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley Longman, Reading, 2nd edition, 1998.

[KV03] Jyrki Katajainen and Fabio Vitale. Navigation piles with applications to sorting, priority queues, and priority deques. *Nordic Journal of Computing*, 10:238–262, 2003.

[LL96] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics*, 1:Article 4, 1996.

[LL99] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31:66–104, 1999.

[NBC+95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4:603–627, 1995.

[San00] Peter Sanders. Fast priority queues for cached memory. *The ACM Journal of Experimental Algorithmics*, 5:Article 7, 2000.

[SP03] Christian Ulrik Søttrup and Jakob Gregor Pedersen. CPH STL's benchmark værktøj. CPH STL Report 2003-1, Department of Computing, University of Copenhagen, 2003. Available at `http://www.cphstl.dk`.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

Appendix

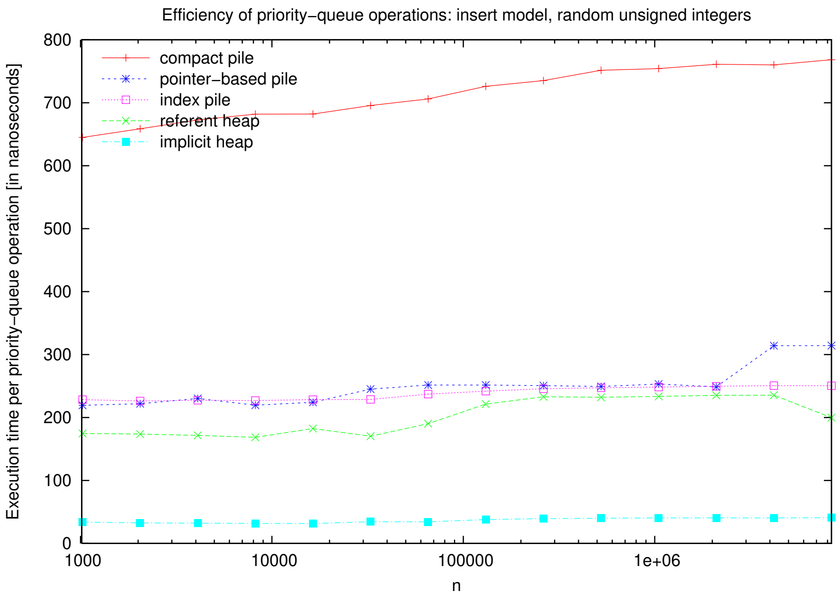

Figure 3.6: Performance of the implementations for the insert model using random unsigned integers on an Intel Pentium 4 workstation.

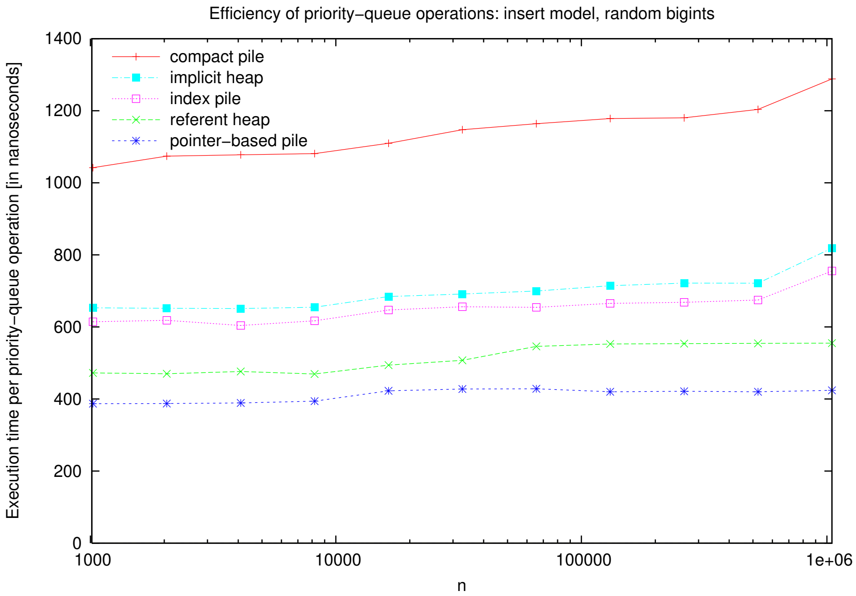

Figure 3.7: Performance of the implementations for the insert model using random bigints on an Intel Pentium 4 workstation.

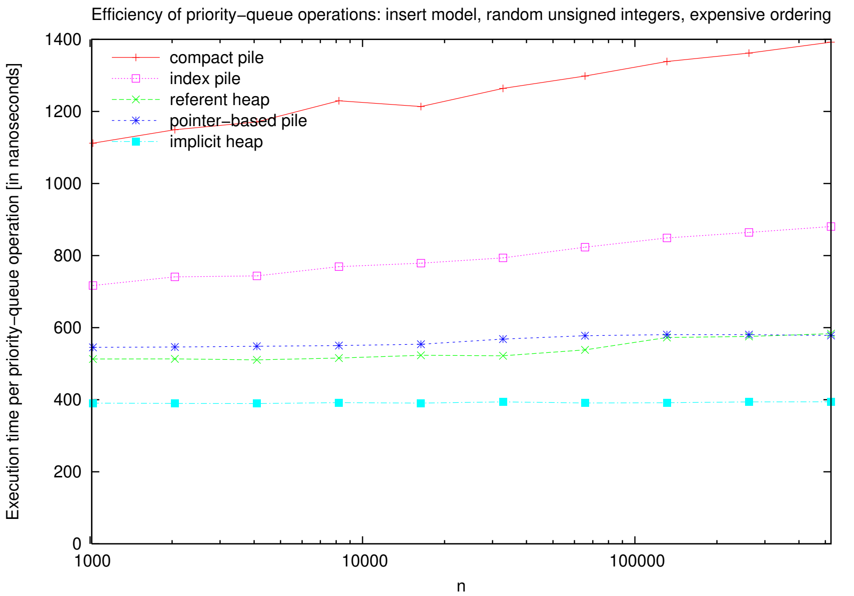

Figure 3.8: Performance of the implementations for the insert model using random unsigned integers with an expensive ordering function on an Intel Pentium 4 workstation.

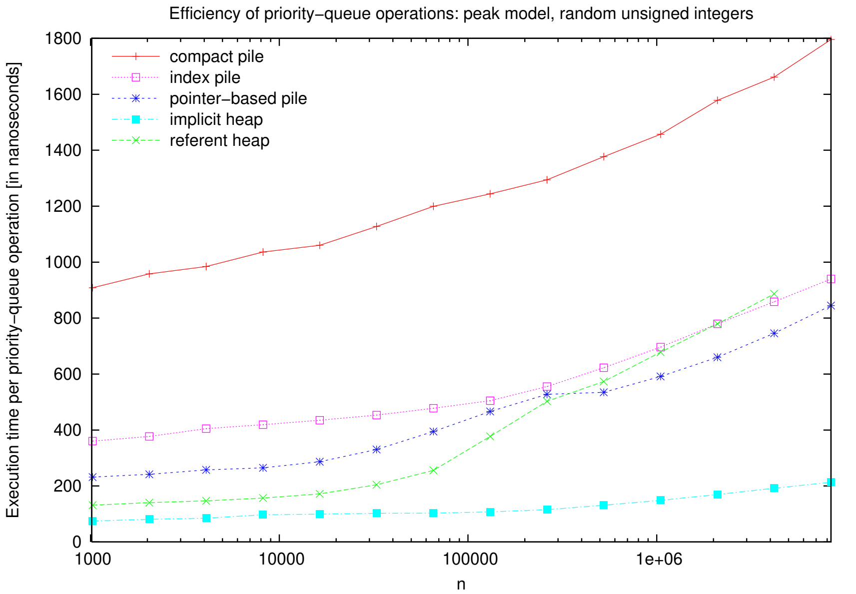

Figure 3.9: Performance of the implementations for the peak model using random unsigned integers on an Intel Pentium 4 workstation.

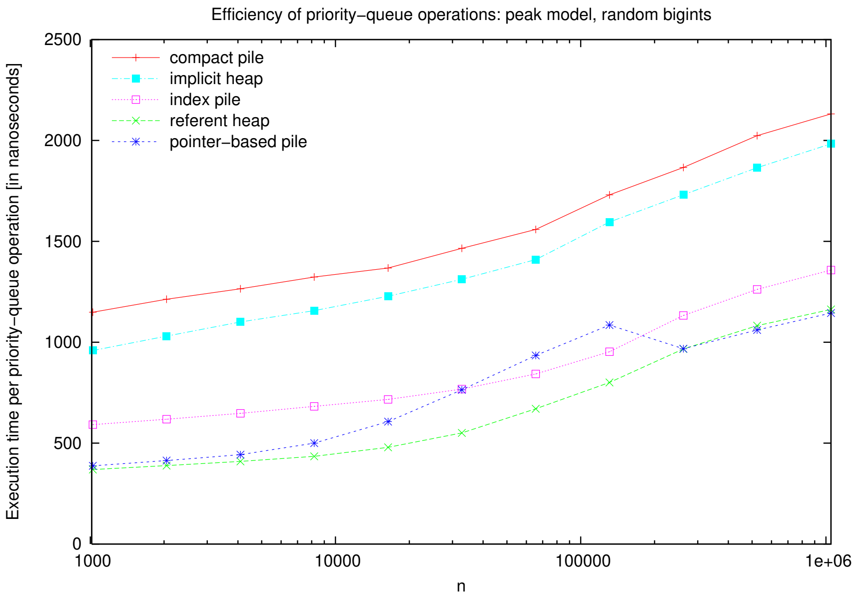

Figure 3.10: Performance of the implementations for the peak model using random bigints on an Intel Pentium 4 workstation.

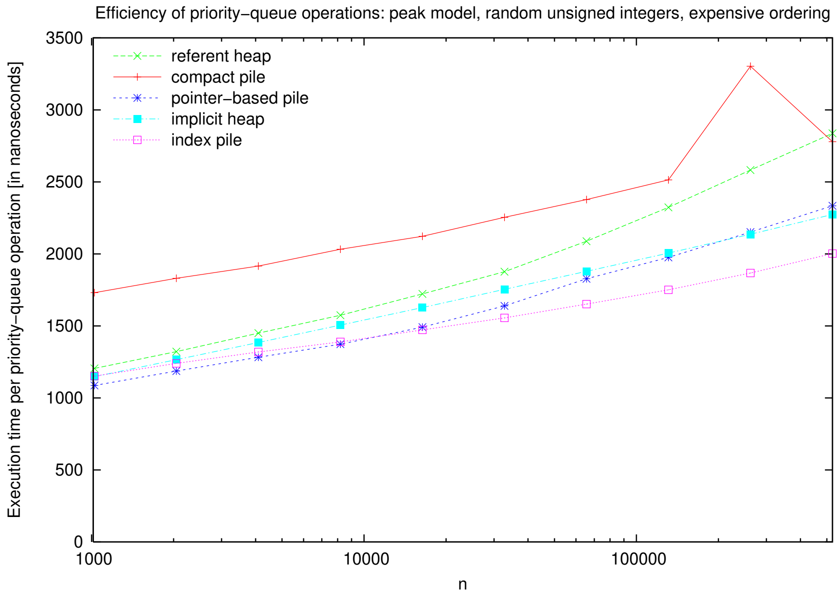

Figure 3.11: Performance of the implementations for the peak model using random unsigned integers with an expensive ordering function on an Intel Pentium 4 workstation.

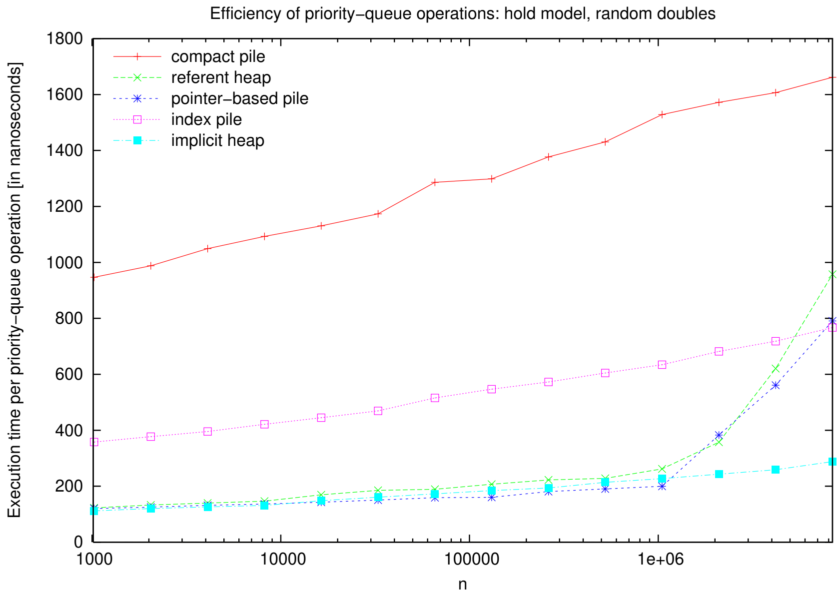

Figure 3.12: Performance of the implementations for the hold model using random doubles on an Intel Pentium 4 workstation.

Chapter 4

A Framework for Speeding Up Priority-Queue Operations

Abstract. We introduce a framework for reducing the number of element comparisons performed in priority-queue operations. In particular, we give a priority queue which guarantees the worst-case cost of $O(1)$ per minimum finding and insertion, and the worst-case cost of $O(\log n)$ with at most $\log n+O(1)$ element comparisons per minimum deletion and deletion, improving the bound of $2\log n+O(1)$ on the number of element comparisons known for binomial queues and pairing heaps. Here, n denotes the number of elements stored in the data structure prior to the operation in question, and $\log n$ equals $\max\{1, \log_2 n\}$. We also give a priority queue that provides, in addition to the above-mentioned methods, the priority-decrease (or decrease-key) method. This priority queue achieves the worst-case cost of $O(1)$ per minimum finding, insertion, and priority decrease; and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log\log n)$ element comparisons per minimum deletion and deletion.

4.1 Introduction

One of the major research issues in the field of theoretical computer science is the comparison complexity of computational problems. In this paper, we consider priority queues (called heaps in some texts) that guarantee a cost of $O(1)$ for insert, with an attempt to reduce the number of element comparisons involved in delete-min. Binary heaps [Wil64] are therefore excluded, following from the fact that $\log\log n \pm O(1)$ element comparisons are necessary and sufficient for inserting an element into a heap of size n [GM86]. Gonnet and Munro [GM86] (corrected by Carlsson [Car91]) also showed that $\log n + \log^* n \pm O(1)$ element comparisons are necessary and sufficient for deleting a minimum element from a binary heap.

In the literature several priority queues have been proposed that achieve a cost of $O(1)$ per find-min and insert, and a cost of $O(\log n)$ per delete-min and delete. Examples of priority queues that achieve these bounds, in the amortized sense [Tar85], are binomial queues [Bro78, Vui78] and pairing heaps [FSST86, Iac00]. The same efficiency can be achieved in the worst case with a special implementation of a binomial queue (see, for example, [CMP88] or [DGST88, Section 3]). If the decrease (often called decrease-key) method is to be supported, Fibonacci heaps [FT87] and thin heaps [KT99] achieve, in the amortized sense, a cost of $O(1)$ per find-min, insert, and decrease; and a cost of $O(\log n)$ per delete-min and delete. Run-relaxed heaps [DGST88], fat heaps [KT99], and meldable priority queues described in [Bro96] achieve these bounds in the worst case.

Among the priority queues that support insertions at a cost of $O(1)$, binomial queues and pairing heaps guarantee that the number of element comparisons performed per delete-min is bounded above by $2\log n + O(1)$. For the other aforementioned priority queues guaranteeing a cost of $O(1)$ per insert, the bound on the number of element comparisons involved in delete-min exceeds $2\log n$.

In all our data structures we use various forms of binomial trees as the basic building blocks. Therefore, in Section 4.2 we review how binomial trees are employed in binomial queues (called binomial heaps in [CLRS01]). In Section 4.3, we present our two-tier framework for structuring priority queues. We apply the framework in three different ways to reduce the number of element comparisons performed in priority-queue operations. In Section 4.8, we discuss which other data structures could be used in our framework as a substitute for binomial trees.

The results of this paper are as follows. In Section 4.4, we give a structure, called a *two-tier binomial queue*, that guarantees the worst-case cost of $O(1)$ per find-min and insert, and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log\log n)$ element comparisons per delete-min and delete. In Section 4.5, we describe a refined priority queue, called a *multipartite binomial queue*, by which the better bound of at most $\log n + O(1)$ element comparisons per delete-min and delete is achieved. In Section 4.6, we show as an application of the framework that, by using a multipartite binomial queue in adaptive heapsort [LP93], a sorting algorithm is obtained that is optimally adaptive with respect to the inversion measure of disorder, and that sorts a sequence having n elements and I inversions with at most $n\log(I/n) + O(n)$ element comparisons. This is the first priority-queue-based sorting algorithm having these properties. Both in Section 4.5 and in Section 4.6 the results presented are stronger than those presented in the conference version of this paper [Elm04].

In Section 4.7, we present a priority queue, called a *multipartite relaxed binomial queue*, that provides the decrease method in addition to the above-mentioned methods. The data structure is built upon run-relaxed binomial queues (called run-relaxed heaps in [DGST88]). A multipartite relaxed binomial queue guarantees the worst-case cost of $O(1)$ per insert, find-min, and decrease; and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log\log n)$ element comparisons per delete-min and delete. During the course of this work we perceived an interesting taxonomy between different building blocks that can be used in our framework. In the conference version of this paper [Elm04], it was outlined that with structures similar to thin binomial trees [KT99] a priority queue is obtained that guarantees, in the amortized sense, a cost of $O(1)$ per insert, find-min, and decrease; and a cost of $O(\log n)$ with at most $1.44 \log n + O(\log \log n)$ element comparisons per delete-min and delete. With fat trees [KT99] the same costs can be achieved in the worst case and the number of element comparisons performed per delete-min and delete can be reduced to $1.27 \log n + O(\log \log n)$. Finally, with relaxed binomial trees [DGST88] the same worst-case bounds are achieved, except that the constant factor in the logarithm term in the bound on the number of element comparisons for delete-min and delete can be reduced to one.

4.2 Binomial queues

In a generic form, a *priority queue* is a data structure which depends on four type parameters: $\mathcal{E}$, $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{A}$. $\mathcal{E}$ is the type of the *elements* manipulated; $\mathcal{C}$ is the type of the *compartments* used for storing the elements, one element per compartment; and $\mathcal{F}$ is the type of the *ordering function* used in element comparisons. A compartment may also contain satellite data, like references to other compartments. We assume that the elements can only be moved and compared, both operations having a cost of $O(1)$. Furthermore, we assume that it is possible to get any datum stored at a compartment at a cost of $O(1)$. $\mathcal{A}$ is the type of the *allocator* which provides methods for allocating new compartments and deallocating old compartments. We omit the details concerning memory management, and simply assume that both allocation and deallocation have a cost of $O(1)$.

Any priority queue $Q\langle\mathcal{E}, \mathcal{C}, \mathcal{F}, \mathcal{A}\rangle$ should provide the following methods:

$\mathcal{E}$ **find-min().** Return a minimum element stored in Q. The minimum is taken with respect to $\mathcal{F}$. **Requirement.** The data structure is not empty. The element returned is passed by reference.

$\mathcal{C}$ **insert($\mathcal{E}$ e).** Insert element e into Q and return its compartment for later use. **Requirement.** There is space available to accomplish this operation. Both e and the returned compartment are passed by reference.

void delete-min(). Remove a minimum element and its compartment from Q. **Requirement.** The data structure is not empty.

void delete($\mathcal{C}$ x). Remove both the element stored at compartment x and compartment x from Q. **Requirement.** x is a valid compartment. x is passed by reference.

Another method that may be considered is:

void decrease($\mathcal{C}$ x, $\mathcal{E}$ e)**.** Replace the element stored at compartment x with element e. **Requirement.** x is a valid compartment. e is no greater than the old element stored at x. Both x and e are passed by reference.

Some additional methods — like a constructor, a destructor, and a set of methods for examining the number of elements stored in Q — are necessary to make the data structure useful, but these are computationally less interesting and therefore not considered here.

We would like to point out that, after inserting an element, the reference to the compartment where it is stored should remain the same so that possible later references made by delete and decrease operations are valid. In some sources this problem is not acknowledged, meaning that the proposed algorithms are actually incorrect. Our solution to this potential problem is simple: we do not move the elements after they have been inserted into the data structure. For other solutions, we refer to a longer discussion in [HR02].

In a tree its *nodes* are used as compartments for storing the elements. A *binomial tree* [Bro78, SPP76, Vui78] is a rooted, ordered tree defined recursively as follows. A binomial tree of rank 0 is a single node. For $r > 0$, a binomial tree of rank r comprises the root and its r binomial subtrees of rank $0, 1, \ldots, r-1$ in this order. We call the root of the subtree of rank 0 the *oldest child* and the root of the subtree of rank $r-1$ the *youngest child*. It follows directly from the definition that the size of a binomial tree is always a power of two, and that the *rank* of a tree of size 2^r is r.

A binomial tree can be implemented using the *child-sibling representation*, where every node has three pointers, one pointing to its youngest child, one to its closest younger sibling, and one to its closest older sibling. The children of a node are kept in a circular, doubly-linked list, called the *child list*, so one of the sibling pointers of the youngest child points to the oldest child, and vice versa. Unused child pointers have the value null. In addition, each node should store the rank of the maximal subtree rooted at it. To facilitate the delete method, every node should have space for a parent pointer, but the parent pointer is set only if the node is the youngest child of its parent. To distinguish the root from the other nodes, its parent pointer is set to point to a fixed sentinel; for other nodes the parent pointer points to another node or has the value null.

The children of a node can be sequentially accessed by traversing the child list from the youngest to the oldest, or vice versa if the oldest child is first accessed via the youngest child. It should be pointed out that with respect to the parent pointers our representation is nonstandard. An argument, why one parent pointer per child list is enough and why we can afford to visit all younger siblings of a node to get to its parent, is given in Lemma 1. In our representation each node has a constant number of pointers pointing to it, and it knows from which nodes those pointers come. Because of this, it is possible to detach any node by updating only a constant number of pointers.

In its standard form, a *binomial queue* is a forest of binomial trees with at most one tree of any given rank. In addition, the trees are kept *heap ordered*, i.e. the element stored at every node is no greater than the elements stored at the children of that node. The sibling pointers of the roots are reused to keep the trees in a circular, doubly-linked list, called the *root list*, where the binomial trees appear in increasing order of rank.

Two binomial trees of the same rank can be linked together by making the root of the tree that stores the greater element the youngest child of the other root. Later on, we refer to this as a *join*. A *split* is the inverse of a join, where the subtree rooted at the youngest child of the root is unlinked from the given tree. A join involves a single element comparison, and both a join and a split have a cost of $O(1)$.

Let B be a binomial queue. The priority-queue operations for B can be implemented as follows:

B.**find-min().** The root storing a minimum element is accessed and that element is returned. The other operations are given the obligation to maintain a pointer to the location of the current minimum.

B.**insert(e).** A new node storing element e is constructed and then added to the forest as a tree of rank 0. If this results in two trees of rank 0, successive joins are performed until no two trees have the same rank. Furthermore, the pointer to the location of the current minimum is updated if necessary.

B.**delete-min().** The root storing an overall minimum element is removed, thus leaving all the subtrees of that node as independent trees. In the set of trees containing the new trees and the previous trees held in the binomial queue, all trees of equal ranks are joined until no two trees of the same rank remain. The root storing a new minimum element is then found by scanning the current roots and the pointer to the location of the current minimum is updated accordingly.

B.**delete(x).** The binomial tree containing node x is traversed upwards starting from x, the current node is swapped with its parent, and this is repeated until the root of the tree is reached. Note carefully that nodes are swapped by detaching them from their corresponding child lists and attaching them back in each others place. Since whole nodes are swapped, pointers to the nodes from the outside remain valid. Lastly, the root is removed as in a delete-min operation.

B.**decrease(x, e).** The element stored at node x is replaced with element e and node x is repeatedly swapped with its parent until the heap order is reestablished. Also, the pointer to the location of the current minimum is updated if necessary.

For a binomial queue storing n elements, the worst-case cost per find-min is $O(1)$ and that per insert, delete-min, delete, and decrease is $O(\log n)$. The amortized bound on the number of element comparisons is two per insert and $2\log n + O(1)$ per delete-min. To see that the bound is tight for delete-min (and delete), consider a binomial queue of size n which is one less than a power of two, an operation sequence which consists of pairs of delete-min and insert, and a situation where the element to be deleted is always stored at the root of the tree of the largest rank. Every delete-min operation in such a sequence needs $\lfloor \log n \rfloor$ element comparisons for joining the trees of equal ranks and $\lfloor \log n \rfloor$ element comparisons for finding the root that stores a new minimum element.

To get the worst-case cost of $O(1)$ for an insert operation, all the necessary joins cannot be performed at once. Instead, a constant number of joins can be done in connection with each insertion, and the execution of the other joins can be delayed for forthcoming insert operations. To facilitate this, a logarithmic number of pointers to joins in process is maintained on a stack. More closely, each pointer points to a root in the root list; the rank of the tree pointed to should be the same as the rank of its neighbour. In one *join step*, the pointer at the top of the stack is popped, the two roots are removed from the root list, the corresponding trees are joined, and the root of the resulting tree is put in the place of the two. If there exists another tree of the same rank as the resulting tree, a pointer indicating this pair is pushed onto the stack. Thereby a preference is given for joins involving small trees.

In an insert operation a new node is created and added to the root list. If the given element is smaller than the current minimum, the pointer indicating the location of a minimum element is updated to point to the newly created node. If there exists another tree of rank 0, a pointer to this pair of trees is pushed on the stack. After this a constant number of join steps is executed. If one join is done in connection with every insert operation, the on-going joins are already disjoint and there are always space for new elements (for a similar treatment, see [CMP88] or [CK77, p. 53 ff.]). Analogously with an observation made in [CMP88], the size of the stack can be reduced dramatically if two join steps are executed in connection with every insert operation, instead of one.

Since there are at most two trees of any given rank, the number of element comparisons performed by a delete-min and delete operation is never larger than $3 \log n$. In fact, a tighter analysis shows that the number of trees is bounded by $\lfloor \log n \rfloor + 1$. The argument is that insert, delete-min, and delete operations can be shown to maintain the invariant that any rank occupying two trees is preceded by a rank occupying no tree, possibly having a sequence of ranks occupying one tree in between. That is, the number of element comparisons is only at most $2 \log n + O(1)$ per delete-min and delete. An alternative way of achieving the worst-case bounds, two element comparisons per insert and $2 \log n + O(1)$ element comparisons per delete-min/delete, is described in [DGST88, Section 3].

4.3 Two-tier framework

For the binomial queues there are two major tasks that contribute to the multiplicative factor of two in the bound on the number of element comparisons for delete-min. The first is the join of trees of equal ranks, and the second is the maintenance of the pointer to the location of a minimum element. The key idea of our framework is to reduce the number of element comparisons involved in finding a new minimum element after the joins.

To realize this idea we compose a priority queue using the following three components, which themselves are priority queues:

1. The *lower store* is a priority queue which stores at least half of all of the n elements. This store is implemented as a collection of separate structures, the size of each of which is an exact power of two. Each element is stored only once, and there is no relation between elements held in different structures. A special requirement for delete-min and delete is that they only modify one of the structures and at the same time retain the size of that structure. In addition to the normal priority-queue methods, *structure borrowing* should be supported in which an arbitrary structure can be released from the lower store (and moved to the reservoir if this becomes empty). As to the complexity requirements, find-min and insert should have a cost of $O(1)$, and delete-min and delete a cost of $O(\log n)$. Moreover, structure borrowing should have a cost of $O(1)$.

2. The *upper store* is a priority queue which stores pointers to the m structures in the lower store, each giving one minimum candidate. In pointer comparisons, the candidates referred to are compared. The main purpose of the upper store is to provide fast access to an overall minimum element in the lower store. The requirement is that find-min and insert should have a cost of $O(1)$, and delete-min and delete a cost of $O(\log m)$.

3. The *reservoir* is a special priority queue which supports find-min, delete-min, and delete, but not insert. It contains the elements that are not in the lower store. Whenever a compartment together with the associated element is deleted from the lower store, as a result of a delete-min or delete operation, a *compartment* is *borrowed* from the reservoir. Using this borrowed compartment, the structure that lost a compartment can be readjusted to gain the same properties as before the deletion. Again, find-min should have a cost of $O(1)$, and delete-min and delete a cost of $O(\log n)$, where n is the number of *all* elements stored. In other words, the cost need only be logarithmic in the size of the reservoir at the time when the reservoir is refilled by borrowing a structure from the lower store. Moreover, compartment borrowing should have a cost of $O(1)$.

To get from the lower store to the upper store and from the upper store to the lower store, we assume that each structure in the lower store is linked to the corresponding pointer in the upper store, and vice versa. Moreover, to distinguish whether a compartment is in the reservoir or not, we assume that each structure has extra information indicating the component in which the structure is held, and that this information can easily be reached from each compartment.

Let I be an implementation-independent framework interface for a priority queue. Using the priority-queue operations provided for the components, the priority-queue operations for I can be realized as follows:

I.**find-min().** A minimum element is either in the lower store or in the reservoir, so it can be found by lower-store find-min — which relies on upper-store find-min — or by reservoir find-min. The smaller of these two elements is returned.

I.**insert(e).** The given element e is inserted into the lower store using lower-store insert, which may invoke the operations provided for the upper store.

I.**delete-min().** First, if the reservoir is empty, a group of elements is moved from the lower store to the reservoir using structure borrowing. Second, lower-store find-min and reservoir find-min are invoked to determine in which component an overall minimum element lies. Depending on the outcome, lower-store delete-min or reservoir delete-min is invoked. If an element is to be removed from the lower store, another element is borrowed from the reservoir to retain the size of the modified lower-store structure. Depending on the changes made in the lower store, it may be necessary to update the upper store as well.

I.**delete(x).** It is first made sure that the reservoir is not empty; if it is, it is refilled by borrowing a structure from the lower store. The extra information, associated with the structure in which the given compartment x is stored, is accessed. If the compartment is in the reservoir, reservoir delete is invoked; otherwise, lower-store delete is invoked. In lower-store delete, a compartment is borrowed from the reservoir to retain the size of the modified structure. If necessary, the upper store is updated as well.

Assume now that the given complexity requirements are fulfilled. Since a lower-store find-min operation and a reservoir find-min operation have a cost of $O(1)$, a find-min operation has a cost of $O(1)$. The efficiency of an insert operation is directly related to that of a lower-store insert operation, i.e. the cost is $O(1)$. In a delete-min operation the cost of the find-min and insert operations invoked is only $O(1)$. Also, compartment borrowing and structure borrowing have a cost of $O(1)$. Let n denote the number of elements stored, and let $D_\ell(n)$, $D_u(n)$, and $D_r(n)$ be the functions expressing the complexity of lower-store delete-min, upper-store delete-min, and reservoir delete-min, respectively. Hence, the complexity of a delete-min operation is bounded above by $\max\{D_\ell(n) + D_u(n), D_r(n)\} + O(1)$. As to the efficiency of a delete operation, there is a similar dependency on the efficiency of lower-store delete, upper-store delete, and reservoir delete. The number of element comparisons performed can be analysed after the actual realization of the components is detailed.

4.4 Two-tier binomial queues

In our first realization of the framework we use binomial trees as the basic structures, and utilize binomial queues in the form described in Section 4.2. Therefore, we call the data structure *two-tier binomial queue*. Its components are the following:

1. The lower store is implemented as a binomial queue storing the major part of the elements.

2. The upper store is implemented as another binomial queue that stores pointers to the roots of the binomial trees held in the lower store, but the upper store may also store pointers to earlier roots of the lower store that are currently either in the reservoir or inner nodes in the lower store.

3. The reservoir consists of a single tree, which is binomial at the time of its creation.

The form of the nodes is identical in the lower store and the reservoir, and each node is linked to the corresponding node in the upper store; if no counterpart in the upper store exists, the link has the value null. Also, we use the convention that the parent pointer of the root of the reservoir points to a reservoir sentinel, whereas for the trees held in the lower store the parent pointers of the roots point to a lower-store sentinel. This way we can easily distinguish the origin of a root. Instead of compartments and structures, nodes and subtrees are borrowed by exchanging references to these objects. We refer to these operations as *node borrowing* and *tree borrowing*.

If there are n elements in total, the size of the upper store is $O(\log n)$. Therefore, at the upper store, delete-min and delete require $O(\log \log n)$ element comparisons. The challenge is to maintain the upper store and to implement the priority-queue operations for the lower store such that the work done in the upper store is reduced. If in the lower store the removal of a root is implemented in the standard way, there might be a logarithmic number of new subtrees that need to be inserted into the upper store. Possibly, some of the new subtrees have to be joined with the existing trees, which again may cascade a high number of deletions to the upper store. Hence, as required, a new implementation of the removal of a root is introduced that alters only one of the lower-store trees.

Next, we show how different priority-queue operations may be handled. We describe and analyse the operations for the reservoir, the upper store, and the lower store in this order.

4.4.1 Reservoir operations

To borrow a node from the tree of the reservoir, the oldest child of the root is detached (or the root itself if it does not have any children), making the children of the detached node the oldest children of the root in the same order. Due to the circularity of the child list, the oldest child and its neighbouring nodes can be accessed by following a few pointers. So the oldest child can be detached from the child list at a cost of $O(1)$. Similarly, two child lists can be appended at a cost of $O(1)$. To sum up, the total cost of node borrowing is $O(1)$.

A find-min operation simply returns the element stored at the root of the tree held in the reservoir. That is, the worst-case cost of a find-min operation is $O(1)$.

In a delete-min operation, the root of the tree of the reservoir is removed and the subtrees rooted at its children are *repeatedly joined* by processing the children of the root from the oldest to the youngest. In other words, every subtree is joined with the tree which results from the joins of the subtrees rooted at the older children. In a delete operation, the given node is repeatedly swapped with its parent until the root is reached, the root is removed, and the subtrees of the removed root are repeatedly joined. In both delete-min and delete, when the removed node has a counterpart in the upper store, the counterpart is deleted as well.

For the analysis, the invariants proved in the following lemmas are crucial. For a node x in a rooted tree, let A_x be the set of ancestors of x, including x itself; let C_x be the number of all the siblings of x that are younger than x, including x; and let D_x be $\sum_{y \in A_x} C_y$.

Lemma 1. *For any node x in a binomial tree of rank r, $D_x \leq r + 1$.*

Proof. The proof is by induction. Clearly, the claim is true for a tree consisting of a single node. Assume that the claim is true for two trees T_1 and T_2 of rank $r - 1$. Without loss of generality, assume that the root of T_2 becomes the root after T_1 and T_2 are joined together. For every node x in T_1, D_x increases by one due to the new root. For every node y in T_2, except the root, D_y increases by one because the only ancestor of y that gets a new younger sibling is the child of the new root. Now the claim follows from the induction assumption. □

Lemma 2. *Consider any node x of the tree held in the reservoir. Starting with a binomial tree of rank r, D_x never gets larger than $r + 1$ during the life-span of this tree.*

Proof. By Lemma 1, the initial tree fulfils the claim. Node borrowing modifies the tree in the reservoir by removing the oldest child of the root and moving all its children one level up. For every node x in any of the subtrees rooted at the children of the oldest child of the root, D_x will decrease by one. For all other nodes the value remains the same. Hence, if the claim was true before borrowing, it must also be true after the modifications.

Each delete-min and delete operation removes the root of the tree in the reservoir and repeatedly joins the resulting subtrees. Due to the removal of the root, for every node x, D_x decreases by one. Moreover, since the subtrees are made separate, if there are j subtrees in all, for any node y in the subtree rooted at the ith oldest child (or simply the ith subtree), $i \in \{1, \ldots, j\}$, D_y decreases by $j - i$. A join increases D_x by one for every node x in the subtrees involved, except that the value remains the same for the root. Therefore, since a node x in the ith subtree is involved in $j - i + 1$ joins, D_x may increase at most by $j - i + 1$. To sum up, for every node x, D_x may only decrease or stay the same. Hence, if the claim was true before the root removal, it must also be valid after all the modifications. □

Corollary 3. *During the life-span of the tree held in the reservoir, starting with a binomial tree of rank r, the root of the tree has at most r children.*

Proof. For a node x, let d_x denote the number of children of x. Let y be the root of the tree held in the reservoir and z the oldest child of y. Clearly, $D_z = d_y + 1$. By Lemma 2, $D_z \leq r + 1$ all the time, and thus, $d_y \leq r$. $\square$

The complexity of a delete-min and delete operation is directly related to the number of children of the root, and the complexity of a delete operation is also related to the length of the D_x-path for the node x being deleted. If the rank of the tree in the reservoir was initially r, by Corollary 3 the number of children of the root is always smaller than or equal to r, and by Lemma 2 the length of the D_x-path is bounded by r. During the life-span of the tree held in the reservoir, there is another binomial tree in the lower store whose rank is at least r (see Section 4.4.3). Thus, if n denotes the number of elements stored, $r < \log n$. The update of the upper store, if at all necessary, has an extra cost of $O(\log \log n)$, including $O(\log \log n)$ element comparisons. Hence, the worst-case cost of a delete-min and delete operation is $O(\log n)$ and the number of element comparisons performed is at most $\log n + O(\log \log n)$.

4.4.2 Upper-store operations

The upper store is a worst-case efficient binomial queue storing pointers to the nodes held in the other two components. In addition to the standard priority-queue methods, it supports lazy deletions where nodes are marked to be deleted instead of being removed immediately. It should also be possible to unmark a node if the node pointed to by the stored pointer becomes a root later on. The invariant maintained by the algorithms is that for each marked node, whose pointer refers to a node y in the lower store or in the reservoir, there is another node x such that the element stored at x is no greater than the element stored at y.

To obtain worst-case efficient lazy deletions, we use the global-rebuilding technique adopted from [OvL81]. When the number of unmarked nodes becomes equal to $m_0/2$, where m_0 is the current size of the upper store, we start building a new upper store. The work is distributed over the forthcoming $m_0/4$ upper-store operations. In spite of the reorganization, both the old structure and the new structure are kept operational and used in parallel. All new nodes are inserted into the new structure, and all old nodes being deleted are removed from their respective structures. Since the old structure does not need to handle insertions, the trees there can be emptied as in the reservoir by detaching the oldest child of the root in question, or the root itself if it does not have any children. If there are several trees left, if possible, a tree whose root does not contain the current minimum is selected as the target of each detachment.

In connection with each of the next at most $m_0/4$ upper-store operations, four nodes are detached from the old structure; if a node is unmarked, it is inserted into the new structure; otherwise, it is released and in its counterpart in the lower store the pointer to the upper store is given the value null. When the old structure becomes empty, it is dismissed and thereafter the new structure is used alone. During the $m_0/4$ operations at most $m_0/4$ nodes can be deleted or marked to be deleted, and since there were $m_0/2$ unmarked nodes in the beginning, at least half of the nodes are unmarked in the new structure. Therefore, at any point in time, we are constructing at most one new structure. We emphasize that each node can only exist in one structure and whole nodes are moved from one structure to the other, so that pointers from the outside remain valid.

Since the cost of each detachment and insertion is $O(1)$, the reorganization only adds an additive term $O(1)$ to the cost of all upper-store operations. A find-min operation, which is a normal binomial-queue operation, may need to consult both the old and the new upper stores, so its worst-case cost is still $O(1)$. The actual cost of marking and unmarking is clearly $O(1)$, even if they take part in reorganizations. If m denotes the total number of unmarked nodes currently stored, at any point in time, the total number of nodes stored is $\Theta(m)$, and during a reorganization $m_0 = \Theta(m)$. According to our earlier analysis, in the old structure the efficiency of delete-min and delete operations depends on the original size m_0. In the new structure their efficiency depends on the current size m. Therefore, since delete-min and delete operations are handled normally, except that they may take part in reorganizations, each of them has the worst-case cost of $O(\log m)$ and may perform at most $2 \log m + O(1)$ element comparisons.

4.4.3 Lower-store operations

A find-min operation simply invokes upper-store find-min and then follows the received pointer to the root storing a minimum element. Clearly, the worst-case cost of a find-min operation is $O(1)$.

An insert operation is accomplished, in a worst-case efficient manner, as described in Section 4.2. As a result of joins, some roots of the trees in the lower store are linked to other roots, so the corresponding pointers should be deleted from the upper store. Instead of using upper-store delete, lazy deletion is applied. The worst-case cost of each join is $O(1)$ and the worst-case cost of each lazy deletion is also $O(1)$. Since each insert operation only performs a constant number of joins and lazy deletions, its worst-case cost is $O(1)$.

Prior to each delete-min and delete operation, it is checked whether a reservoir refill is necessary. If the reservoir is empty, a tree of the highest rank is taken from the lower store. If the tree is of rank 0, it is moved to the reservoir and the corresponding pointer is deleted from the upper store. This special case when $n = 1$ can be handled at a cost of $O(1)$. In the normal case, the tree taken is split into two halves, and the subtree rooted at the youngest child is moved to the reservoir. The other half is kept in the lower store. However, if after the split the lower store contains another tree of the same rank as the remaining half, the two trees are joined and the pointer to the root of the loser tree is to be deleted from the upper store. Again, lazy deletion is applied. A join has a cost of $O(1)$ and involves one element comparison. As shown, each lazy deletion has a cost of $O(1)$, including also some element comparisons. That is, the total cost of tree borrowing is $O(1)$.

In a delete-min operation, after a possible reservoir refill the root storing a minimum element is removed, a node from the reservoir is borrowed, and the borrowed node — seen as a tree of rank 0 — is repeatedly joined with the subtrees of the removed root. This results in a new binomial tree with the same structure as before the deletion. In the upper store, a pointer to the new root of the resulting tree is inserted and the pointer to the old root is deleted. However, if the pointer to the root already exists in the upper store, the upper-store node containing that pointer is simply unmarked. In a delete operation, after a possible reservoir refill the given node is swapped to the root as in a delete operation for a binomial queue, after which the root is deleted as in a delete-min operation.

As analysed earlier, tree borrowing and node borrowing have the worst-case cost of $O(1)$. Also, the removal of a root has the worst-case cost of $O(1)$. The at most $\lfloor \log n \rfloor$ joins executed have the worst-case cost of $O(\log n)$, and the number of element comparisons performed is at most $\log n$. The upper-store update has an additional cost of $O(\log \log n)$, including $O(\log \log n)$ element comparisons. To summarize, the worst-case cost of a delete-min operation is $O(\log n)$ and the number of element comparisons performed is at most $\log n + O(\log \log n)$. As to a delete operation, since in a binomial tree of size n the length of any D_x-path is never longer than $\log n$, node swapping has the worst-case cost of $O(\log n)$, but involves no element comparisons. Therefore, the complexity of a delete operation is the same as that of a delete-min operation.

4.4.4 Summing up the results

Using the components described and the complexity bounds derived, the efficiency of the priority-queue operations supported by the framework interface can be summed up as follows:

Theorem 4. *Let n be the number of elements stored in the data structure prior to each priority-queue operation. A two-tier binomial queue guarantees the worst-case cost of $O(1)$ per find-min and insert, and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log \log n)$ element comparisons per delete-min and delete.*

The bound on the number of element comparisons for delete-min and delete can be further reduced. Instead of having two levels of priority queues, we can have several levels. At each level, except the highest one, delete-min and delete operations are carried out as in our earlier lower store relying on a reservoir; and at each level, except the lowest one, lazy deletions are carried out as in our earlier upper store. Except for the highest level, the constant factor in the logarithm term expressing the number of element comparisons performed per delete-min or delete is one. Therefore the total number of element comparisons performed in all levels is at most $\log n + \log \log n + \ldots + O(\log^{(k)} n)$, where $\log^{(k)}$

denotes the logarithm function applied k times and k is a constant representing the number of levels. An insertion of a new element would result in a constant number of insertions and lazy deletions per level. Hence, the number of levels should be a fixed constant to achieve a constant cost for insertions.

4.5 Multipartite binomial queues

In this section we present a refinement of a two-tier binomial queue, called a *multipartite binomial queue*. To refine the previous construction, the following modifications are significant:

1. The lower store is divided into three components: main store, insert buffer, and floating tree. The *main store* is maintained as the lower store in our earlier construction. However, all insert operations are directed to the *insert buffer* which is a binomial queue maintained in a worst-case efficient manner. When the insert buffer becomes too large, a subtree is cut off from one of its trees and used as a cutting for the *floating tree*. The floating tree is *incrementally united* with the existing trees in the main store in connection with each modifying operation. That is, joins make the floating tree larger, and when uniting is complete, the floating tree becomes part of the main store. At any point in time, we ensure that the size of the insert buffer is logarithmic in the total number of elements stored. We also ensure that uniting will be finished before it will be necessary to create a new floating tree.

2. The upper store is implemented as a circular, doubly-linked list; there is one node per tree held in the main store. Each node contains a pointer to the root of the tree which stores a minimum element among the elements stored in the trees having a lower rank, including the tree itself. We call these pointers the *prefix-minimum pointers*.

3. A reservoir is still in use and all the reservoir operations are performed as previously described, but to refill it a subtree is borrowed from the insert buffer and if the insert buffer is empty from the main store. Borrowing is never necessary from the floating tree since during the whole uniting process the insert buffer will be large enough to service the refills which may be needed.

The nodes in the main store, insert buffer, floating tree, and reservoir may be moved from one component to another, so the form of the nodes must be identical in all four components.

In the improved construction the key idea is to balance the work done in the main store and the upper store. After using $r + O(1)$ element comparisons to readjust a tree of rank r in the main store, only $\log n - r + O(1)$ element comparisons are used for the maintenance of the upper store. Another important idea is to unite the floating tree to the main store such that all the involved components, the main store, insert buffer, floating tree, and reservoir, are fully operational during the uniting operation.

4.5.1 Description of the components

An upper-store find-min operation provides a minimum element in the main store by following the prefix-minimum pointer for the tree of the highest rank. Thus, a find-min operation has the worst-case cost of $O(1)$. To delete a pointer corresponding to a tree of rank r from the upper store, the node in question is found by a sequential scan and thereafter removed, and the prefix-minimum pointers are updated for all trees having a rank higher than r. The total cost is proportional to $\log n$ and one element comparison per higher-rank tree is necessary, meaning at most $\log n - r + O(1)$ element comparisons. When the element stored at the root of a tree of rank r is changed, the prefix-minimum pointers can be updated in a similar manner. The complexity of such a change is the same as that of a delete operation. To insert a pointer corresponding to a tree of rank r, as done in the uniting process, a sequential scan has to be done to find the correct insertion point. It may happen that there already exists a tree of the same rank. Therefore, these trees must be joined and this join may propagate to all the higher ranks. In addition, the prefix-minimum pointers must be updated, but this can be done simultaneously with the joins, if at all necessary, so that each of the higher-rank trees is considered only once. Hence, the worst-case cost of an insert operation is proportional to $\log n$, and at most $2(\log n - r) + O(1)$ element comparisons are performed.

The main store is a binomial queue that is maintained as our earlier lower store, except that the main store and the upper store interact in another way. Tree borrowing is done almost as before. First, the tree of the highest rank is taken from the main store, the tree is split, one half of it is moved to the reservoir, and the other half is kept in the main store; the latter half is then joined with another tree of the same rank if there is any. Second, the prefix-minimum pointers for the trees of the two highest ranks are updated. The total cost of all these modifications is $O(1)$. From this and our earlier analysis, it follows that the worst-case cost of tree borrowing is $O(1)$.

Since no normal insertions are done in the main store, no lazy deletions are forwarded to the upper store. Because of node borrowing, only one tree need to be modified in a delete-min or delete operation. If the rank of the modified tree is r, both main-store operations have the worst-case cost of $O(r)$ and require at most $r + O(1)$ element comparisons. After the adjustment in the main store, the prefix-minimum pointers need to be updated in the upper store for all trees having a rank higher than or equal to r. This has an additional cost proportional to $\log n$, and $\log n - r + O(1)$ element comparisons may be necessary. To summarize, the worst-case cost of a delete-min and delete operation is $O(\log n)$ and never more than $\log n + O(1)$ element comparisons are performed.

The insert buffer is implemented as a worst-case efficient binomial queue. Hence, an insert operation has the worst-case cost of $O(1)$. To reduce the size of the insert buffer or to refill the reservoir, tree borrowing is used as for the main store. Observe that after tree borrowing no change to the pointer indicating the location of the current minimum of the insert buffer is necessary. As will be shown, the invariants maintained guarantee that the insert buffer will never become larger than $c_1 \log n + c_2$ for some constants c_1 and c_2. Therefore, a delete-min and delete operation has the worst-case cost of $O(\log \log n)$ and performs at most $2\log(c_1 \log n + c_2) + O(1)$ element comparisons, which is bounded by $\log n + O(1)$ for all $n \geq 0$.

The floating tree is maintained as the trees in our earlier lower store. The main point is that delete-min and delete operations should retain the size of this tree. Therefore, when a root is removed, a node from the reservoir is borrowed. Our earlier analysis implies that the worst-case cost of a delete-min and delete operation is $O(\log n)$, and the number of element comparisons performed is at most $\log n + O(1)$.

4.5.2 Interactions between the components

We let the priority-queue operations change the data structure in *phases*. Let n_0 denote the total number of elements at the beginning of a phase. All operations are made aware of the current phase using n_0, $\lfloor \log n_0 \rfloor$, and a single counter. To avoid the usage of the whole-number logarithm function, $\lfloor \log n_0 \rfloor$ can be calculated by maintaining the interval $[2^k \,..\, 2^{k+1})$ in which n_0 lies. When n_0 moves outside the interval, the logarithm and the interval are updated accordingly.

When performing the priority-queue operations the following invariants are maintained:

1. In a phase, exactly $\lfloor \log n_0 \rfloor$ *modifying* operations — insert, delete-min, or delete — are executed.

2. The number of elements in the floating tree is no smaller than $\log n_0$ if the tree exists, i.e. at least $\log n_0$ elements are extracted from the insert buffer if an extraction is done.

3. At the beginning of the phase, the insert buffer contains no more than $\max\{24, 9\log n_0\}$ elements, i.e. the insert buffer never gets too large.

4. At the beginning of the phase, there is no floating tree, i.e. the floating tree from the previous phase, if any, has been successfully united to the main store.

The first invariant is forced by the protocol used for handling the priority-queue operations. Initially, the other invariants are valid since all the components are empty. The first and third invariants imply that, for all $n \geq 0$, the insert buffer never gets larger than $c_1 \log n + c_2$ for some constants c_1 and c_2.

At the beginning of a phase, the first modifying operation executes a *preprocessing step* prior to its actual task in order to make the insert buffer smaller, if necessary. If the size of the insert buffer is larger than $8\log n_0$, half of a tree of the highest rank is borrowed and used to form a new floating tree. On the other hand, if the size of the insert buffer is smaller than or equal to $8\log n_0$, no changes are made to the insert buffer and no floating tree is created. Next we analyse the consequences of the preprocessing step.

Let us assume that, when a tree is borrowed in the preprocessing step, in the insert buffer a tree of the highest rank is of size 2^k. This tree would be the smallest possible if, for all $i \in \{0, 1, \ldots, k\}$, there existed two trees of size 2^i. Then the number of elements stored in the insert buffer would be $2^{k+2} - 2$. Since in the insert buffer there are at least $\lceil 8 \log n_0 \rceil - 1$ elements, $\lceil 8 \log n_0 \rceil - 1 \leq 2^{k+2} - 2$, from which it follows that $\lceil \log n_0 \rceil \leq 2^{k-1}$. Since the size of the borrowed tree is 2^{k-1}, the size of the floating tree must be at least $\lceil \log n_0 \rceil$ at the time of its creation. During the uniting process, the floating tree can only become larger, so the second invariant is established.

Assume that a phase involves i_0 insertions, $0 \leq i_0 \leq \lfloor \log n_0 \rfloor$, and d_0 deletions, $0 \leq d_0 \leq \lfloor \log n_0 \rfloor$. Let b_0 denote the number of elements stored in the insert buffer at the beginning of the phase, and n_1 the total number of elements at the end of the phase. To analyse the effect of the preprocessing step on the size of the insert buffer, we consider five cases:

Case 1. $n_0 \leq 24$. Since $b_0 \leq n_0$, $b_0 < 8 \log n_0$ for all $0 \leq b_0 \leq n_0 \leq 24$. Thus, $b_0 + \lfloor \log n_0 \rfloor < 9 \log n_0$. If $n_1 > n_0$, we are done. Otherwise, if $n_1 \leq n_0$, then $n_1 \leq 24$ and hence the size of the insert buffer is less than 24 at the end of the phase.

Case 2. $n_0 > 24$, $i_0 \geq d_0$, and $b_0 \leq 8 \log n_0$. Since $i_0 \geq d_0$, $n_1 \geq n_0$. If $b_0 \leq 8 \log n_0$, the size of the insert buffer must be bounded by $9 \log n_0 \leq 9 \log n_1$ at the end of the phase.

Case 3. $n_0 > 24$, $i_0 \geq d_0$, and $b_0 > 8 \log n_0$. Since $b_0 > 8 \log n_0$, at least $\log n_0$ elements must have been extracted from the insert buffer in the preprocessing step. Therefore, at the end of the phase the insert buffer cannot be larger than b_0. Since $i_0 \geq d_0$, it must be that $n_1 \geq n_0$. So if $b_0 \leq 9 \log n_0$ at the beginning of the phase, the size of the insert buffer must be bounded by $9 \log n_1$ at the end of the phase.

Case 4. $n_0 > 24$, $i_0 < d_0$, and $b_0 \leq 8 \log n_0$. Since $i_0 < d_0$, at most half of the modifying operations have been insertions. Moreover, it must be true that $n_1 \geq n_0 - \lfloor \log n_0 \rfloor$. Since $8.5 \log n_0 \leq 9 \log(n_0 - \lfloor \log n_0 \rfloor)$ for all $n_0 > 24$, the insert buffer cannot be larger than $9 \log n_1$ at the end of the phase. The above-mentioned inequality is easy to verify for n_0 larger than 2^{18}. We used a computer to verify it for all integers in the range $\{25, 26, \ldots, 2^{18}\}$.

Case 5. $n_0 > 24$, $i_0 < d_0$, and $b_0 > 8 \log n_0$. Again, since $i_0 < d_0$, at most half of the modifying operations involved in the phase have been insertions. Since $b_0 > 8 \log n_0$, at least $\log n_0$ elements must have been extracted from the insert buffer in the preprocessing step. Thus, the insert buffer cannot be larger than $b_0 - \log n_0 + (1/2)\lfloor \log n_0 \rfloor$ at the end of the phase. This means that its size must be bounded by $8.5 \log n_0$. As in Case 4,

$n_1 \geq n_0 - \lfloor \log n_0 \rfloor$. So by the same argument as in Case 4, the input buffer cannot be larger than $9 \log n_1$ at the end of the phase.

In particular, note that reservoir refills only make the insert buffer smaller, so these cannot cause any harm. In conclusion, if the insert buffer was not larger than $\max\{24, 9 \log n_0\}$ at the beginning of the phase, it cannot be larger than $\max\{24, 9 \log n_1\}$ at the end of the phase. Thus, the third invariant is established.

Basically, to unite the floating tree and the main store a normal insert operation for binomial queues is executed, except that the insertion starts from a rank higher than 0. In the worst case, uniting may involve logarithmically many joins so it is done incrementally. This means that the prefix-minimum pointers are not necessarily valid for trees whose rank is higher than the rank of the tree up to which the uniting process has advanced. To solve the problem, each find-min operation should consult two trees; one referred to by the prefix-minimum pointer for the tree up to which the uniting process has advanced and the other referred to by the prefix-minimum pointer for the tree of the highest rank.

The main store can have at most $\lceil \log(n_0 - \lfloor 8 \log n_0 \rfloor) \rceil$ trees when a new floating tree is created, so this is the highest rank before uniting. Of course, the trees having a rank lower than the rank of the given tree can be skipped. Since $\lceil \log(n_0 - \lfloor 8 \log n_0 \rfloor) \rceil \leq \lfloor \log n_0 \rfloor$ for all positive n_0, at most one tree need to be visited in connection with each modifying operation. At each visit, one join step is executed and the corresponding prefix-minimum pointer updated. At this speed, the uniting process will be finished before the end of the phase is reached, which establishes the fourth invariant.

Now that we have proved the correctness of the invariants, we can analyse the efficiency of the priority-queue operations. The overhead caused by the phase management and the preprocessing step is only $O(1)$ per modifying operation. Also, incremental uniting will only increase the cost of modifying operations by an additive constant.

In a find-min operation, the four components storing elements — main store, reservoir, floating tree, and insert buffer — need to be consulted. Since all these components support a find-min operation at the worst-case cost of $O(1)$, the worst-case cost of a find-min operation is $O(1)$. Insertions only involve the insert buffer so, from the bound derived for worst-case efficient binomial queues and the fact that the extra overhead per insert is $O(1)$, the worst-case cost of $O(1)$ directly follows.

Each delete-min operation refills the reservoir if necessary, determines in which component an overall minimum element is stored, and thereafter invokes the corresponding delete-min operation provided for that component. According to our earlier analysis, each of the components storing elements supports a delete-min operation at the worst-case cost of $O(\log n)$, including at most $\log n + O(1)$ element comparisons. Even with other overheads, the bounds are the same.

In a delete operation, the root is consulted to determine which of the delete operations provided for the components storing elements should be invoked. The traversal to the root has the worst-case cost of $O(\log n)$, but even with this and other overheads, a delete operation has the worst-case cost of $O(\log n)$ and performs at most $\log n + O(1)$ element comparisons as shown earlier.

To conclude, we have proved the following theorem.

Theorem 5. *Let n be the number of elements stored in the data structure prior to each priority-queue operation. A multipartite binomial queue guarantees the worst-case cost of $O(1)$ per find-min and insert, and the worst-case cost of $O(\log n)$ with at most $\log n + O(1)$ element comparisons per delete-min and delete.*

4.6 Application: adaptive heapsort

A sorting algorithm is adaptive if it can sort all input sequences and performs particularly well for sequences having a high degree of existing order. The cost consumed is allowed to increase with the amount of disorder in the input. In the literature many adaptive sorting algorithms have been proposed and many measures of disorder considered (for a survey, see [ECW92] or [MP92]). In this section we consider adaptive heapsort, introduced by Levcopoulos and Petersson [LP93], which is one of the simplest adaptive sorting algorithms. As in [LP93], we assume that all input elements are distinct.

At the commencement of adaptive heapsort a Cartesian tree is built from the input sequence. Given a sequence $X = \langle x_1, \ldots, x_n \rangle$, the corresponding *Cartesian tree* [Vui80] is a binary tree whose root stores element $x_i = \min\{x_1, \ldots, x_n\}$, the left subtree of the root is the Cartesian tree for sequence $\langle x_1, \ldots, x_{i-1} \rangle$, and the right subtree is the Cartesian tree for sequence $\langle x_{i+1}, \ldots, x_n \rangle$. After building the Cartesian tree, a priority queue is initialized by inserting the element stored at the root of the Cartesian tree into it. In each of the following n iterations, a minimum element stored in the priority queue is output and thereafter deleted, the elements stored at the children of the node that contained the deleted element are retrieved from the Cartesian tree, and the retrieved elements are inserted into the priority queue.

The total cost of the algorithm is dominated by the cost of the n insertions and n minimum deletions; the cost involved in building [GBT84] and querying the Cartesian tree is linear. The basic idea of the algorithm is that only those elements that can be the minimum of the remaining elements are kept in the priority queue, not all elements. Levcopoulos and Petersson [LP93] showed that, when element x_i is deleted, the number of elements in the priority queue is no greater than $\lfloor |Cross(x_i)|/2 \rfloor + 2$, where

$$Cross(x_i) = \big\{ j \mid j \in \{1, \ldots, n\} \text{ and } \min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\} \big\}.$$

Levcopoulos and Petersson [LP93, Corollary 20] showed that adaptive heapsort is optimally adaptive with respect to *Osc*, *Inv*, and several other measures of disorder. For a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of length n, the measures *Osc* and *Inv* are defined as follows:

$$Osc(X) = \sum_{i=1}^{n} |\mathit{Cross}(x_i)|$$

$$\mathit{Inv}(X) = \big|\{(i,j) \mid i \in \{1,2,\ldots,n-1\},\, j \in \{i+1,\ldots,n\}, \text{ and } x_i > x_j\}\big|.$$

The optimality with respect to the *Inv* measure, which measures the number of pairs of elements that are in wrong order, follows from the fact that $Osc(X) \leq 4\mathit{Inv}(X)$ for any sequence X [LP93].

Implicitly, Levcopoulos and Petersson showed that using an advanced implementation of binary-heap operations the cost of adaptive heapsort is proportional to

$$\sum_{i=1}^{n} \big(\log |\mathit{Cross}(x_i)| + 2\log\log |\mathit{Cross}(x_i)|\big) + O(n)$$

and that this is an upper bound on the number of element comparisons performed. Using a multipartite binomial queue, instead of a binary heap, we get rid of the $\log\log$ term and achieve the bound

$$\sum_{i=1}^{n} \log |\mathit{Cross}(x_i)| + O(n).$$

Because the geometric mean is never larger than the arithmetic mean, it follows that our version is optimally adaptive with respect to the measure *Osc*, and performs no more than $n \log\big(Osc(X)/n\big) + O(n)$ element comparisons when sorting a sequence X of length n. From this, the bounds for the measure *Inv* immediately follow: the cost is $O\big(n \log\big(\mathit{Inv}(X)/n\big)\big)$ and the number of element comparisons performed is $n \log\big(\mathit{Inv}(X)/n\big) + O(n)$. Other adaptive sorting algorithms that guarantee the same bounds are either based on insertionsort or mergesort [EF03].

4.7 Multipartite relaxed binomial queues

In this section, our main goal is to extend the repertoire of priority-queue methods to include the decrease method. There are two alternative ways of relaxing the definition of a binomial queue to support a fast decrease operation. In run-relaxed heaps [DGST88], heap-order violations are allowed and a separate structure is maintained to keep track of all violations. In Fibonacci heaps [FT87] and thin heaps [KT99], structural violations are allowed; in general, some nodes may have lost some of their children. We tried both approaches; with the former approach we were able to achieve better bounds even though for the best realizations in both categories the difference was only in the lower-order terms.

We use relaxed binomial trees, as defined in [DGST88], as our basic building blocks. Our third priority queue has multiple components and the interactions between the components are similar to those in a multipartite binomial queue.

The main difference is that there is no separate reservoir, but the insert buffer is kept nonempty so it can support node borrowing. Since all components are implemented as run-relaxed binomial queues [DGST88] with some minor variations, we call the resulting data structure a *multipartite relaxed binomial queue*. We describe the data structure in three parts. First, we recall the details of run-relaxed binomial queues, but we still assume that the reader is familiar with the original paper by Driscoll et al. [DGST88], where the data structure was introduced. Second, we show how the upper store is maintained. Third, we explain how the insert buffer and the main store are organized.

The following theorem summarizes the main result of this section.

Theorem 6. *Let n be the number of elements stored in the data structure prior to each priority-queue operation. A multipartite relaxed binomial queue guarantees the worst-case cost of $O(1)$ per find-min, insert, and decrease, and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log\log n)$ element comparisons per delete-min and delete.*

4.7.1 Run-relaxed binomial queues

A *relaxed binomial tree* [DGST88] is an almost heap-ordered binomial tree where some nodes are denoted to be *active*, indicating that the element stored at that node may be smaller than the element stored at the parent of that node. Nodes are made active by a decrease operation if the replaced element causes a heap-order violation between the accessed node and its parent. Even though a later priority-queue operation may repair the heap-order violation, the node can still be active. A node remains active until the heap-order violation is explicitly removed. From the definition, it directly follows that a root cannot be active. A *singleton* is an active node whose immediate siblings are not active. A *run* is a maximal sequence of two or more active nodes that are consecutive siblings.

Let τ denote the number of trees in any collection of relaxed binomial trees, and let λ denote the number of active nodes in these trees, i.e. in the *entire* collection of trees. A *run-relaxed binomial queue* (called a run-relaxed heap in [DGST88]) is a collection of relaxed binomial trees where $\tau \leq \lfloor \log n \rfloor + 1$ and $\lambda \leq \lfloor \log n \rfloor$, n denoting the number of elements stored.

To keep track of the trees in a run-relaxed binomial queue, the roots are doubly linked together as in a binomial queue. To keep track of the active nodes, a *run-singleton structure* is maintained as described in [DGST88]. All singletons are kept in a *singleton table*, which is a resizable array accessed by rank. In particular, this table must be implemented in such a way that growing and shrinking at the tail is possible at the worst-case cost of $O(1)$, which is achievable, for example, by doubling, halving, and incremental copying (see also [BCD$^+$99, KM01]). Singletons of the same rank are kept in a list. Each entry of the singleton table has a counterpart in a *pair list* if there are more than one singleton of that rank. The youngest active node of each run is kept in a *run list*. All lists are doubly linked, and each active node should have a pointer to its occurrence in a list (if any). The bookkeeping details are quite straightforward so we will not repeat them here, but refer to [DGST88]. The fundamental operations supported are an addition of a new active node, a removal of a given active node, and a removal of at least one arbitrary active node if λ is larger than $\lfloor \log n \rfloor$. The cost of each of these operations is $O(1)$ in the worst case.

As to the transformations needed for reducing the number of active nodes, we again refer to the original description given in [DGST88]. The rationale behind the transformations is that, when there are more than $\lfloor \log n \rfloor$ active nodes, there must be at least one pair of active nodes that root a subtree of the same rank or there is a run of two or more neighbouring active nodes. In that case, it is possible to apply at least one of the transformations — singleton transformations or run transformations — to reduce the number of active nodes by at least one. The cost of performing any of the transformations is $O(1)$ in the worst case. Later on, one application of the transformations together with all necessary changes to the run-singleton structure is referred to as a λ-*reduction*.

Each tree in a run-relaxed binomial queue can be represented in the same way as a normal binomial tree, but to support the transformations used for reducing the number of active nodes some additional data need to be stored at the nodes. In addition to sibling pointers, a child pointer, and a rank, each node should contain a pointer to its parent and a pointer to its occurrence in the run-singleton structure. The occurrence pointer of every nonactive node has the value null; for a node that is active and in a run, but not the last in the run, the pointer is set to point to a fixed run sentinel; and for all other nodes the pointer gives the occurrence in the run-singleton structure. To support our framework, each node should store yet another pointer to its counterpart in the upper store, and vice versa.

Let us now consider how the priority-queue methods can be implemented. A reader familiar with the original paper by Driscoll et al. [DGST88] should be aware that we have made some minor modifications to the find-min, insert, delete-min, and delete methods to adapt them for our purposes.

A minimum element can be stored at one of the roots or at one of the active nodes. To facilitate a fast find-min operation, a pointer to the node storing a minimum element is maintained. When such a pointer is available, a find-min operation can be accomplished at the worst-case cost of $O(1)$.

An insert operation is performed in the same way as in a worst-case efficient binomial queue. As pointed out in Section 4.2, even if some of the joins are delayed, there can never be more than $\lfloor \log n \rfloor + 1$ trees. From our earlier analysis, it follows that an insert operation has the worst-case cost of $O(1)$ and requires at most two element comparisons.

In delete-min and delete operations, we rely on the same borrowing technique as in [DGST88]: the root of a tree of the smallest rank is borrowed to fill in the hole created by the node being removed. To free a node that can be borrowed, a tree of the smallest rank is repeatedly split, if necessary, until the split results in a tree of rank 0. In one *split step*, if x denotes the root of a tree of the smallest rank and y its youngest child, the tree rooted at x is split, and if y is active, it is made nonactive and its occurrence is removed from the run-singleton structure. Note that this splitting does not have any effect on the pointer indicating the location of a minimum element, since no nodes are removed.

A delete-min operation has two cases depending on whether one of the roots or one of the active nodes is to be removed. Similarly, a delete operation has two cases depending on whether the given node is a root or not. Next, we consider the two forms of deletions, deletion of a root and deletion of one of the inner nodes, separately.

Let z denote the node being deleted, and assume that z is a root. If the tree rooted at z has rank 0, z is simply removed and no other structural changes are done. Otherwise, the tree rooted at z is repeatedly split and, when the tree rooted at z has rank 0, z is removed. Compared to above, each split step is modified such that all active children of z are retained active, but they are temporarily removed from the run-singleton structure (since the structure of runs may change). Thereafter, the freed tree of rank 0 and the subtrees rooted at the children of z are repeatedly joined by processing the trees in increasing order of rank. Finally, the active nodes temporarily removed are added back to the run-singleton structure. The resulting tree replaces the tree rooted at z in the root list. It would be possible to handle the tree used for borrowing and the tree rooted at z symmetrically, with respect to treating the active nodes, but when the delete-min/delete method is embedded into our two-tier framework, it would be too expensive to remove all active children of z in the course of a single delete-min/delete operation.

To complete the operation, all roots and active nodes are scanned to update the pointer indicating the location of a minimum element. Singletons are found by scanning through all lists in the singleton table. Runs are found by accessing the youngest nodes via the run list and for each such node by following the sibling pointers until a nonactive node is reached.

The computational cost of a delete-min/delete operation, when a root is being deleted, is dominated by the repeated splits, the repeated joins, and the scan over all minimum candidates. In each of these steps a logarithmic number of nodes is visited so the total cost of these operations is $O(\log n)$. Splits as well as updates to the run-singleton structure do not involve any element comparisons. In total, joins may involve at most $\lfloor \log n \rfloor$ element comparisons. Even though a tree of the smallest rank is split, after the joins the number of trees is at most $\lfloor \log n \rfloor + 1$. Since no new active nodes are created, the number of active nodes is still at most $\lfloor \log n \rfloor$. To find the minimum of $2\lfloor \log n \rfloor + 1$ elements, at most $2\lfloor \log n \rfloor$ element comparisons are to be done. To summarize, this form of a delete-min/delete operation performs at most $3 \log n$ element comparisons.

Assume now that the node z being deleted is an inner node, and let x be the node borrowed. Also in this case the tree rooted at z is repeatedly split, and after removing z the tree of rank 0 rooted at x and the subtrees of the children of z are repeatedly joined. The resulting tree is put in the place of the subtree rooted earlier at z. If z was active and contained the current minimum, the operation is completed by updating the pointer to the location of a minimum element. If x is the root of the resulting subtree and a heap-order violation is introduced, node x is made active and the number of active nodes is reduced by one, if necessary, by performing a single λ-reduction.

Similar to the case of deleting a root, this case has the worst-case cost of $O(\log n)$. If z did not contain the current minimum, only at most $\lfloor \log n \rfloor + O(1)$ element comparisons are done; at most $\lfloor \log n \rfloor$ due to joins and $O(1)$ due to a λ-reduction. However, if z contained the current minimum, at most $2\lfloor \log n \rfloor$ additional element comparisons may be necessary. That is, the total number of element comparisons performed is bounded by $3 \log n + O(1)$. To sum up, each delete-min/delete operation has the worst-case cost of $O(\log n)$ and performs at most $3 \log n + O(1)$ element comparisons.

A decrease operation is performed as in [DGST88]. After making the element replacement, it is checked whether the replacement causes a heap-order violation between the given node and its parent. If there is no violation, the operation is complete. Otherwise, the given node is made active, an occurrence is inserted into the run-singleton structure, and a single λ-reduction is performed if the number of active nodes is larger than $\lfloor \log n \rfloor$. If the given element is smaller than the current minimum, the pointer indicating the location of a minimum element is corrected to point to the given node. All these modifications have the worst-case cost of $O(1)$.

4.7.2 Upper-store operations

The upper store contains pointers to the roots of the trees held in the insert buffer and in the main store, pointers to all active nodes in the insert buffer and in the main store, and pointers to some earlier roots and active nodes. The number of trees in the insert buffer is at most $\log \log n + O(1)$, the number of trees in the main store is at most $\lfloor \log n \rfloor + 1$, and the number of active nodes is at most $\lfloor \log n \rfloor$. The last property follows from the fact that the insert buffer and the main store share the same run-singleton structure. At any given point in time only a constant fraction of the nodes in the upper store can be marked to be deleted. Hence, the number of pointers is $O(\log n)$.

The upper store is implemented as a run-relaxed binomial queue. In addition to the priority-queue methods find-min, insert, delete-min, delete, and decrease, which are realized as described earlier, it should be possible to mark nodes to be deleted and to unmark nodes if they reappear at the upper store before being deleted. Lazy deletions are necessary at the upper store when, in the insert buffer or in the main store, a join is done or an active node is made nonactive by a λ-reduction. In both situations, a normal upper-store deletion would be too expensive.

As in Section 4.4, global rebuilding will be used to get rid of the marked nodes when there are too many of them, but for three reasons our earlier procedure is not applicable for run-relaxed binomial queues:

1. Due to parent pointers the oldest child of a root cannot necessarily be detached at a cost of $O(1)$, since the parent pointers of the children of the detached node must be updated as well.

2. The transformations used for reducing the number of active nodes require that the rank of a node and that of its sibling are consecutive. To keep the

old data structure operational, the binomial structure of the trees should not be broken.

3. The transformations might constantly swap subtrees of the same size, so it would be difficult to assure that all nodes have been visited if a simple tree traversal was done incrementally.

Our solution to these problems is repeated splitting. In one *rebuilding step*, if there is no tree of rank 0, a tree of the smallest rank is split into two halves; also if there is only one tree of rank 0 that contains the current minimum, but that is not the only tree left in the old structure, a tree of the smallest rank is split into two halves; otherwise, a tree of rank 0 — other than a tree of rank 0 which contains the current minimum — is removed from the old structure and, if not marked to be deleted, inserted into the new structure. That is, there can simultaneously be three trees of rank 0. This is done in order to keep the pointer to the location of the current minimum valid. As in a delete-min/delete operation, in one split step the youngest child of the root is made nonactive if it is active and its occurrence is removed from the run-singleton structure. With this strategy, a tree of size m can be emptied by performing $2m - 1$ rebuilding steps. Observe also that this strategy is in harmony with the strategy used in delete-min/delete operations; in the old structure the splits made by these operations will only speed up the rebuilding process.

Assume that there are m_0 pointers in the upper store when rebuilding is initiated, and assume that $m_0/2$ of them are marked to be deleted. Rebuilding is done piecewise over the forthcoming $m_0/4$ upper-store operations. More precisely, in connection with each of the following $m_0/4$ upper-store operations eight rebuilding steps are executed. At this speed, even with intermixed upper-store operations, the whole old structure will be empty before it will be necessary to rebuild the new structure.

A tree of rank 0, which does not contain the current minimum or is the only tree left, can be detached from the old run-relaxed binomial queue at a cost of $O(1)$. Similarly, a node can be inserted into the new run-relaxed binomial queue at a cost of $O(1)$. A marked node can also be released and its counterpart updated at a cost of $O(1)$. Also, a split step has the worst-case cost of $O(1)$. From these observations and our earlier analysis, it follows that rebuilding only adds an additive term $O(1)$ to the cost of all upper-store operations.

4.7.3 Insert-buffer and main-store operations

The elements are stored in the insert buffer and in the main store. Both components are implemented as run-relaxed binomial queues, but they have a common run-singleton structure. In the main store there can only be one tree per rank, except perhaps a single rank that may have two trees. All insertions are directed to the insert buffer, which also provides the nodes borrowed by insert-buffer and main-store deletions. Minimum finding relies on the upper store; an overall minimum element is either in one of the roots or in one of the active nodes, stored either in the insert buffer or in the main store. The counterparts of the minimum candidates are stored in the upper-store, so communication between the components storing elements and the upper store is necessary each time when a root or an active node is added or removed, but not when an active node is made a root.

As in a multipartite binomial queue, the priority-queue operations are executed in phases. If n_0 denotes the number of elements at the beginning of a phase, in one phase $\lfloor \log n_0 \rfloor$ modifying operations are carried out. Compared to our earlier construction, the only difference is one additional invariant:

5. At the beginning of the phase, if $n_0 > 0$, the input buffer contains at least $\lfloor \log n_0 \rfloor$ elements, i.e. borrowing is always possible even if all modifying operations in the phase were deletions.

Consider now node borrowing, and assume that the new invariant can be maintained. Since the insert buffer never becomes empty, it has always at least one tree and a tree of the smallest rank can be repeatedly split as prior to a run-relaxed-binomial-queue deletion, after which there is a free tree of rank 0 that can be borrowed. In each split step, if the youngest child of the root of the tree being split is active, it is removed from the run-singleton structure, but it need not be removed from the upper store. However, if the youngest child is not active, its counterpart has to be inserted into the upper store or unmarked if already present in the upper store. Since the size of the insert buffer is bounded by $c_1 \log n + c_2$ for some constants c_1 and c_2, the total cost of all splits is $O(\log \log n)$; and because of the upper-store operations $O(\log \log n)$ element comparisons may be necessary.

Let b_0 denote the size of the insert buffer at the beginning of a phase. To maintain the new invariant, we modify the preprocessing step such that, if $b_0 \leq 2\lfloor \log n_0 \rfloor$, an *incremental separating process* is initiated, the purpose of which is to move a small tree from the main store to the insert buffer. In this process the trees in the main store are visited one by one, starting from the tree of the highest rank, until the smallest tree, the size of which is larger than $2\lfloor \log n_0 \rfloor$, is found. Thereafter, this tree is repeatedly split until a tree is obtained whose size is between $2\lfloor \log n_0 \rfloor$ and $4\lfloor \log n_0 \rfloor$. This work is distributed such that one modifying operation handles one rank. The last operation in such an operation sequence moves a tree of the required size to the insert buffer in its proper place. In the insert buffer there can be at most a constant number of trees that have a higher rank, so this addition has a constant cost, i.e. it is not too expensive for a single modifying operation. If no tree of size $2\lfloor \log n_0 \rfloor$ or larger exists, which is possible when $n_0 \leq 24$, a single operation moves all trees from the main store to the insert buffer and performs all necessary joins.

Even if all modifying operations in a phase were insertions, at the end of the phase the size of the insert buffer would be bounded above by $2\lfloor \log n_0 \rfloor + 4\lfloor \log n_0 \rfloor + \lfloor \log n_0 \rfloor \leq 7\lfloor \log n_0 \rfloor$ or, if $n_0 \leq 24$, by $24 + \lfloor \log n_0 \rfloor$, i.e. the new tree cannot make the insert buffer too large. If all modifying operations were deletions, the size of the insert buffer would be bounded below by $\lfloor \log n_0 \rfloor +$

$2\lfloor \log n_0 \rfloor - \lfloor \log n_0 \rfloor \geq 2\lfloor \log n_0 \rfloor$, i.e. the insert buffer cannot become too small either. Note that there can only be one active uniting process, which is initiated if $b_0 \geq 8\lfloor \log n_0 \rfloor$, or one active separating process, which is initiated if $b_0 \leq 2\lfloor \log n_0 \rfloor$, but not both at the same time. When in an active uniting process a join is done, a lazy deletion is necessary at the upper store; and when in an active separating process a split is done, an insertion may be necessary at the upper store. Hence, these incremental processes can only increase the cost of modifying operations by an additive constant.

An insert operation described for a run-relaxed binomial queue requires two modifications in places where communication between the insert buffer and upper store is necessary. First, after the creation of a new node its counterpart must be added to the upper store. Second, in each join the counterpart of the loser tree must be lazily deleted from the upper store. Even after these modifications, the worst-case cost of an insert operation is $O(1)$.

In a decrease operation, three modifications will be necessary. First, each time when a new active node is created, an insert operation has to be done at the upper store. Second, each time when an active node is removed, the counterpart must be deleted from the upper store, which can be done lazily in a λ-reduction. Third, when the node accessed is a root or an active node, a decrease operation has to be invoked at the upper store. If an active node is made into a root, no change at the upper store is required. Even after these modifications, the worst-case cost of a decrease operation is $O(1)$.

A delete-min/delete operation always begins with an invocation of the procedure that frees a tree of rank 0 to be used for filling in the hole created by the node being deleted. The two forms of deletions are done otherwise as described for a run-relaxed binomial queue, but now the update of the pointer to the location of a minimum element can be avoided. A removal of a root or an active node will invoke a delete operation at the upper store, and an insertion of a new root or an active node will invoke an insert operation at the upper store. A λ-reduction may invoke one or two lazy deletions and at most one insertion at the upper store. These lazy deletions and insertions have the worst-case cost of $O(1)$. Node borrowing has the worst-case cost of $O(\log \log n)$, including $O(\log \log n)$ element comparisons. Only at most one real upper-store deletion will be necessary, which has the worst-case cost of $O(\log \log n)$ and includes $O(\log \log n)$ element comparisons. Therefore, as in the original form, a delete-min/delete operation has the worst-case cost of $O(\log n)$, but now the number of element comparisons performed is at most $\log n + O(\log \log n)$.

4.8 Concluding remarks

We provided a general framework for improving the efficiency of priority-queue operations with respect to the number of element comparisons performed. Essentially, we showed that it is possible to get below the $2 \log n$ barrier on the number of element comparisons performed per delete-min and delete, while keeping the cost of find-min and insert constant. We showed that this is possible even when a decrease operation is to be supported at the worst-case cost of $O(1)$. From the information-theoretic lower bound for sorting, it follows that the worst-case efficiency of insert and delete-min cannot be improved much. However, if the worst-case cost of find-min, insert, and decrease is required to be $O(1)$, we do not know whether the worst-case bound of $\log n + O(\log\log n)$ on the number of element comparisons performed per delete-min and delete could be improved.

The primitives, on which our framework relies, are tree joining, tree splitting, lazy deleting, and node borrowing; all of which have the worst-case cost of $O(1)$. However, as already indicated in Section 4.7, it is not strictly necessary to support so efficient node borrowing. It would be enough if this operation had the worst-case cost of $O(\log n)$, but included no more than $O(1)$ element comparisons. All our priority queues could be implemented, without affecting the complexity bounds derived, to use this weak version of node borrowing.

We used binomial trees as the basic building blocks in our priority queues. The main drawback of binomial trees is their high space consumption. Each node should store four pointers and a rank, in addition to the elements themselves. Assuming that a pointer and an integer can be stored in one word, a multipartite binomial queue uses $5n + O(\log n)$ words, in addition to the n elements. However, if the child list is doubly linked, but not circular, and if the unused pointer to the younger sibling is reused as a parent pointer as in [KT99], weak node borrowing can still be supported, keeping the efficiency of all other fundamental primitives the same. Therefore, if the above-mentioned modification relying on weak node borrowing is used, the space bound could be improved to $4n + O(\log n)$. In order to support lazy deleting, one extra pointer per node is needed, so a two-tier binomial queue requires additional $n + O(\log n)$ words of storage. A multipartite relaxed binomial queue needs even more space, $7n + O(\log n)$ words. As proposed in [DGST88], the space requirement could be reduced by letting each node store a resizable array of pointers to its children.

These space bounds should be compared to the bound achievable for a dynamic binary heap which can be realized using $\Theta(\sqrt{n})$ extra space [BCD+99, KM01]. However, a dynamic binary heap does not keep external references valid and, therefore, cannot support delete or decrease operations. To keep external references valid, a heap could store pointers to the elements instead, and the elements could point back to the respective nodes in the heap. Each time a pointer in the heap is moved, the corresponding pointer from the element to the heap should be updated as well. The references from the outside can refer to the elements themselves which are not moved. With this modification, the space consumption would be $2n + O(\sqrt{n})$ words. Recall, however, that a binary heap cannot support insertions at a cost of $O(1)$.

A navigation pile, proposed by Katajainen and Vitale [KV03], supports weak node borrowing (cf. the second-ancestor technique described in the original paper). All external references can be kept valid if the compartments of the elements are kept fixed, the leaves store pointers to the elements, and the elements point back to the leaves. Furthermore, if pointers are used for expressing parent-child relationships, tree joining and tree splitting become easy. With the above-mentioned modification relying on weak node borrowing, pointer-based navigation piles could substitute for binomial trees in our framework. A navigation pile is a binary tree and, thus, three parent-child pointers per node are required. With the standard trick (see, e.g. [Tar83, Section 4.1]), where the parent and children pointers are made circular, only two pointers per node are needed to indicate parent-child relationships. Taking into account the single pointer stored at each branch and the two additional pointers to keep external references valid, the total space consumption would be $5n + O(\log n)$ words.

It would be interesting to see which data structure performs best in practice when external references to compartments inside the data structure are to be supported. In particular, which data structure should be used when developing an industry-strength priority queue for a program library. It is too early to make any firm conclusions whether our framework would be useful for such a task. To unravel the practical utility of our framework, further investigations would be necessary.

In this paper we studied the comparison complexity of priority-queue operations. A similar question with respect to the number of element comparisons required by all dictionary operations was answered in the affirmative to be $\log n + O(1)$ by Andersson and Lai [AL90]. Still, the trees of Andersson and Lai achieve this bound only in the amortized sense. The existence of a dictionary guaranteeing this bound in the worst case for all dictionary operations is an open problem.

Acknowledgements

We thank Fabio Vitale for reporting the observation, which he made independently of us, that prefix-minimum pointers can be used to speed up delete-min operations for navigation piles.

References

[AL90] Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121, Berlin/Heidelberg, 1990. Springer-Verlag.

[BCD+99] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 37–48, Berlin/Heidelberg, 1999. Springer-Verlag.

[Bro78] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7:298–319, 1978.

[Bro96] Gerth S. Brodal. Worst-case efficient priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58. ACM/SIAM, 1996.

[Car91] Svante Carlsson. An optimal algorithm for deleting the root of a heap. *Information Processing Letters*, 37:117–120, 1991.

[CK77] Michael J. Clancy and Donald E. Knuth. A programming and problem-solving seminar. Technical Report STAN-CS 77-606, Department of Computer Science, Stanford University, 1977.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, 2nd edition, 2001.

[CMP88] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13, Berlin/Heidelberg, 1988. Springer-Verlag.

[DGST88] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computations. *Communications of the ACM*, 31:1343–1354, 1988.

[ECW92] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24:441–476, 1992.

[EF03] Amr Elmasry and Michael L. Fredman. Adaptive sorting and the information theoretic lower bound. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2607 of *Lecture Notes in Computer Science*, pages 654–662, Berlin/Heidelberg, 2003. Springer-Verlag.

[Elm04] Amr Elmasry. Layered heaps. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 212–222, Berlin/Heidelberg, 2004. Springer-Verlag.

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[GBT84] Harold N. Gabow, Jon L. Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 135–143. ACM, 1984.

[GM86] Gaston H. Gonnet and J. Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15:964–971, 1986.

[HR02] Torben Hagerup and Rajeev Raman. An efficient quasidictionary. In *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory*, volume 2368 of *Lecture Notes in Computer Science*, pages 1–18, Berlin/Heidelberg, 2002. Springer-Verlag.

[Iac00] John Iacono. Improved upper bounds for pairing heaps. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 32–45, Berlin/Heidelberg, 2000. Springer-Verlag.

[KM01] Jyrki Katajainen and Bjarke Buur Mortensen. Experiences with the design and implementation of space-efficient deques. In *Proceedings of the 5th Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes in Computer Science*, pages 39–50, Berlin/Heidelberg, 2001. Springer-Verlag.

[KT99] Haim Kaplan and Robert E. Tarjan. New heap data structures. Technical Report TR 597-99, Department of Computer Science, Princeton University, 1999.

[KV03] Jyrki Katajainen and Fabio Vitale. Navigation piles with applications to sorting, priority queues, and priority deques. *Nordic Journal of Computing*, 10:238–262, 2003.

[LP93] Christos Levcopoulos and Ola Petersson. Adaptive heapsort. *Journal of Algorithms*, 14:395–413, 1993.

[MP92] Alistair Moffat and Ola Petersson. An overview of adaptive sorting. *Australian Computer Journal*, 24:70–77, 1992.

[OvL81] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12:168–173, 1981.

[SPP76] Arnold Schönhage, Mike Paterson, and Nicholas Pippenger. Finding the median. *Journal of Computer and Systems Sciences*, 13:184–199, 1976.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–315, 1978.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23:229–239, 1980.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

Chapter 5

Conclusion

A study of the theoretical and practical efficiency of priority queues was presented and some conclusions can be drawn on the basis of the presented work.

The experimental evaluation of in-place d-ary heap showed that no single heapifying strategy has the best performance for the different types of inputs and ordering functions used, but that bottom-up heapifying has a good performance for many of the used inputs and ordering functions.

The experiments with navigation piles showed that when element moves are expensive they can be an alternative to binary heaps. It was possible to replace the second-ancestor technique used in [KV03] with the more simple first-ancestor technique, and it was also possible to transform a static navigation pile into a dynamic navigation pile in a new and simpler way than the one described in the original paper.

It was possible to create a framework for reducing the number of element comparisons performed in priority-queue operations. The framework gives a priority queue which guarantees the worst-case cost of $O(1)$ per *find-min* and *insert*, and the worst-case cost of $O(\log n)$ with at most $\log n + O(1)$ element comparisons per *delete-min* and *delete*. Here, n denotes the number of elements stored in the data structure prior to the operation in question, and $\log n$ equals $\max\{1, \log_2 n\}$. Furthermore, a priority queue that provides *decrease* (also called *decrease-key*), in addition to the above-mentioned operations, was created. This priority queue achieves the worst-case cost of $O(1)$ per *find-min*, *insert*, and *decrease*; and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log \log n)$ element comparisons per *delete-min* and *delete*.

5.1 Further work

For navigation piles the following topics could be of interest: 1) To extend the pointer-based navigation pile to support the *delete* operation (it is possible for pointer-based navigation piles to support *delete* because it guarantees referential integrity); 2) To use the pointer-based implementation of navigation piles to create a data structure supporting *insert* at a worst-case cost of $O(1)$; 3) To implement a dynamic navigation pile using the new and simpler way of making a static navigation pile dynamic.

Furthermore, it could be interesting to explore the practical efficiency of the priority queues described in the paper "A framework for speeding up priority-queue operations" through an experimental study.

5.2 Related work

In connection with the study of the efficiency of priority queues other interesting subjects were encountered, this resulted in the following reports (which are all available through the CPH STL website [Dep06]):

Amr Elmasry, Claus Jensen, and Jyrki Katajainen. Relaxed weak queues: an alternative to run-relaxed heaps. CPH STL Report 2005-2.

Amr Elmasry, Claus Jensen, and Jyrki Katajainen. On the power of structural violations in priority queues. CPH STL Report 2005-3.

Claus Jensen, Jyrki Katajainen, and Fabio Vitale. Experimental evaluation of local heaps. CPH STL Report 2006-1.

Bibliography

[AL90] Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121, Berlin/Heidelberg, 1990. Springer-Verlag.

[Ale01a] Andrei Alexandrescu. Generic<programming>: A policy-based `basic_string` implementation. *C++ Expert Forum*, 2001. Available at `http://www.cuj.com/experts/1096/toc.htm`.

[Ale01b] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Pattern Applied*. Addison-Wesley, Upper Saddle River, 2001.

[BCD+99] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 37–48, Berlin/Heidelberg, 1999. Springer-Verlag.

[BKS00] Jesper Bojesen, Jyrki Katajainen, and Maz Spork. Performance engineering case study: heap construction. *The ACM Journal of Experimental Algorithmics*, 5:Article 15, 2000.

[BM00] Dov Bulka and David Mayhew. *Efficient C++: Performance Programming Techniques*. Addison-Wesley, Reading, 2000.

[Boj98] Jesper Bojesen. Heap implementations and variations. Written project, Department of Computing, University of Copenhagen, Copenhagen, 1998. Available at `http://www.diku.dk/forskning/performance-engineering/Perfeng/resources.html`.

[Bri03] *The C++ Standard: Incorporating Technical Corrigendum 1*. John Wiley and Sons, Ltd., 2nd edition, 2003.

[Bro78] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7:298–319, 1978.

[Bro96] Gerth S. Brodal. Worst-case efficient priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58. ACM/SIAM, 1996.

[BY76] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5:82–87, 1976.

[Car87] Svante Carlsson. A variant of Heapsort with almost optimal number of comparisons. *Information Processing Letters*, 24:247–250, 1987.

[Car91] Svante Carlsson. An optimal algorithm for deleting the root of a heap. *Information Processing Letters*, 37:117–120, 1991.

[Car92] Svante Carlsson. A note on Heapsort. *The Computer Journal*, 35:410–411, 1992.

[CK77] Michael J. Clancy and Donald E. Knuth. A programming and problem-solving seminar. Technical Report STAN-CS 77-606, Department of Computer Science, Stanford University, 1977.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, 2nd edition, 2001.

[CMP88] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13, Berlin/Heidelberg, 1988. Springer-Verlag.

[Dep06] Department of Computing, University of Copenhagen. The CPH STL. Website accessible at `http://www.cphstl.dk/`, 2000–2006.

[DGST88] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computations. *Communications of the ACM*, 31:1343–1354, 1988.

[ECW92] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24:441–476, 1992.

[EF03] Amr Elmasry and Michael L. Fredman. Adaptive sorting and the information theoretic lower bound. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2607 of *Lecture Notes in Computer Science*, pages 654–662, Berlin/Heidelberg, 2003. Springer-Verlag.

[Elm04] Amr Elmasry. Layered heaps. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 212–222, Berlin/Heidelberg, 2004. Springer-Verlag.

[ES02] Stefan Edelkamp and Patrick Stiegeler. Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ comparisons. *The ACM Journal of Experimental Algorithmics*, 7:Article 5, 2002.

[Flo64] Robert W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7:701, 1964.

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[GBT84] Harold N. Gabow, Jon L. Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 135–143. ACM, 1984.

[GM86] Gaston H. Gonnet and J. Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15:964–971, 1986.

[GZ96] Xunrang Gu and Yuzhang Zhu. Optimal heapsort algorithm. *Theoretical Computer Science*, 163:239–243, 1996.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

[HR02] Torben Hagerup and Rajeev Raman. An efficient quasidictionary. In *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory*, volume 2368 of *Lecture Notes in Computer Science*, pages 1–18, Berlin/Heidelberg, 2002. Springer-Verlag.

[Iac00] John Iacono. Improved upper bounds for pairing heaps. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 32–45, Berlin/Heidelberg, 2000. Springer-Verlag.

[Jen01] Brian S. Jensen. Priority queue and heap functions. CPH STL Report 2001-3, Department of Computing, University of Copenhagen, Copenhagen, 2001. Available at `http://www.cphstl.dk/`.

[Joh75] Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4:53–57, 1975.

[Jon86] Douglas W. Jones. An emprirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:300–311, 1986.

[KM01] Jyrki Katajainen and Bjarke Buur Mortensen. Experiences with the design and implementation of space-efficient deques. In *Proceedings of the 5th Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes in Computer Science*, pages 39–50, Berlin/Heidelberg, 2001. Springer-Verlag.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley Longman, Reading, 2nd edition, 1998.

[KPT96] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3:27–40, 1996.

[KT99] Haim Kaplan and Robert E. Tarjan. New heap data structures. Technical Report TR 597-99, Department of Computer Science, Princeton University, 1999.

[KV03] Jyrki Katajainen and Fabio Vitale. Navigation piles with applications to sorting, priority queues, and priority deques. *Nordic Journal of Computing*, 10:238–262, 2003.

[LL96] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics*, 1:Article 4, 1996.

[LL99] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31:66–104, 1999.

[LP93] Christos Levcopoulos and Ola Petersson. Adaptive heapsort. *Journal of Algorithms*, 14:395–413, 1993.

[MP92] Alistair Moffat and Ola Petersson. An overview of adaptive sorting. *Australian Computer Journal*, 24:70–77, 1992.

[Mus97] David R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27:983–993, 1997.

[NBC+95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4:603–627, 1995.

[Oko80] Seiichi Okoma. Generalized Heapsort. In *Proceedings of the 9th Symposium on Mathematical Foundations of Computer Science*, volume 88 of *Lecture Notes in Computer Science*, pages 439–451, Berlin/Heidelberg, 1980. Springer-Verlag.

[OvL81] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12:168–173, 1981.

[Pas] Tomi A. Pasanen. Public communication, December 2003.

[San00] Peter Sanders. Fast priority queues for cached memory. *The ACM Journal of Experimental Algorithmics*, 5:Article 7, 2000.

[Sil04] Silicon Graphics, Inc. Standard template library programmer's guide. Website accessible at `http://www.sgi.com/tech/stl/`, 1993–2004.

[SLK03] Jakob Sloth, Morten Lemvigh, and Mads Kristensen. Sortering i CPH STL. CPH STL Report 2003-2, Department of Computing, University of Copenhagen, Copenhagen, 2003. Available at `http://www.cphstl.dk/`.

[SP03] Christian Ulrik Søttrup and Jakob Gregor Pedersen. CPH STL's benchmark værktøj. CPH STL Report 2003-1, Department of Computing, University of Copenhagen, 2003. Available at `http://www.cphstl.dk`.

[SPP76] Arnold Schönhage, Mike Paterson, and Nicholas Pippenger. Finding the median. *Journal of Computer and Systems Sciences*, 13:184–199, 1976.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–315, 1978.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23:229–239, 1980.

[Weg93] Ingo Wegener. Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). *Theoretical Computer Science*, 118:81–98, 1993.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[WT89] L. M. Wegner and J. I. Teuhola. The external heapsort. *IEEE Transactions on Software Engineering*, 15:917–925, 1989.

Appendix A

Source code

The source code is included on a CD-ROM which is a supplement to this M. Sc. thesis (the source code is also available for download through the homepage of the Performance Engineering Laboratory at `http://www.diku.dk/forskning/performance-engineering/Perfeng/theses.html`). Source code which is not written in connection with this thesis is included on the CD-ROM, this is done in order to make it possible to carry out the tests and benchmarks again, although this may not be possible in other environments than the ones where the tests and benchmarks originally were carried out. The source code not written in connection with this thesis consists of the benchmark tool Benz, a small part of the Boost library from which the static $\log_2$ function is used and the source code related to [San00]. The source code for the implemented algorithms as well as the program tests and some benchmark drivers are written in the C++ program language, the benchmarks themselves are written in the Python program language. The following is a description of where to find the source code located on the CD-ROM.

A.1 Source code associated with Chapter 2

The folders contain the source code for seven heap implementations. The seven implementations are contained in the folder Program which can be found following the path:

CPHSTL/Report/ In-place-multiway-heaps/

This is a list of the names of the implementations used in the Chapter 2 and the folder names:

Top-down basic: "Top-down"
Top-down one-sided binary search: "Binary-search"
Bottom-up basic: "Bottom-up"
Bottom-up two levels at a time: "Two-Level-Bottom-up"
Bottom-up binary search: "Bottom-up-Binary-search"
Bottom-up exponential binary search: "Exp-Binary-search"

Bottom-up move saving: “Bottom-up_move-saving”

A.2 Source code associated with Chapter 3

The folders contain the source code for three navigation-pile implementations and one heap implementation. The four implementations are contained in the folder Programs which can be found following the path:
CPHSTL/Report/Experimental-Navigation-piles/
This is a list of the names of the implementations used in the Chapter 3 and the folder names:
Compact pile: “Packed_Index_based_Navigation_piles”
Index pile: “Index_based_Navigation_piles”
Pointer-based pile: “All_Pointer_based_Navigation_piles”
Referent heap: “Binary_heap_using_reference”

The compact pile implementation uses a resizable bitarray as a utility function, the source code for implementation of the resizable bitarray can be found in the following folder:
CPHSTL/Program/Resizable_bitarray