UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
TESI DI LAUREA SPECIALISTICA IN INFORMATICA

An investigation into efficient priority queues

Relatore: Prof. Paolo Massazza

Correlatore: Prof. Jyrki Katajainen

Candidato: Fabio Vitale
Matr. n. 612293

Anno Accademico 2004/05

Ai miei genitori

Acknowledgements

I would like to thank Prof. Paolo Massazza, my first algorithm professor, for the supervision of the final drafting of the thesis and his precious suggestions; I also would like to express my gratitude to Prof. Jyrki Katajainen, coauthor of the two articles presented in the thesis, for what he taught me, his constant encouragements and all the time he dedicated to our collaboration.

To conclude, I wish to thank my parents for constantly supporting me throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations and goal of the thesis

One of the fundamental aspects that is necessary to consider in the design of an algorithm is the choice of suitable data structures for representing input data, since in most cases the algorithm efficiency strongly depends on the underlying data structures. As a matter of fact, most results associated with the study of performances of algorithms and data structures deal with the asymptotic complexity analysis which require complex analysis techniques.

In the last years, several factors have caused possible deviations between foreseen and real performances. Among these elements there are architecture details of the computer like the hierarchic structure of the memory, the employing of pipeline techniques and the efficiency in the use of registers.

While the results obtained through asymptotic analysis are of primary interest from a theoretical point of view, in many practical situations there is the need of mathematical techniques which take into account real computer architectures (with respect to a suitable level of detail of hardware components).

What is however important to note is that the trend of analysing algorithms and data structures uniquely from the point of view of few significant operations (for example, the number of element comparisons and element movements for sorting algorithms, as a function of the number of elements in input) has lead to a independent research area. The goal of minimizing the number of significant operations represents an interesting and engrossing (at times intriguingly complicated) study independently of real applications.

An example regards the long-standing open problem (proposed for example in [29]) of the research of an in-place sorting algorithm performing (in the worst case) $O(n \log n)$ instructions, $O(n \log n)$ element comparisons and $O(n)$ element moves, solved after years [14] with a solution that does not seem to have many practical applications.

This kind of trend has developed concurrently with the research of data structures and algorithms leaded by the goal to be practically useful. Sometimes the two research lines interlace giving rise to solutions which are significant both for theoretical aspects and the applicative potential.

This last "kind" of algorithms is that one which obviously is generally considered of major interest in all the computer science fields and it therefore represents the main objective, often not completed reached, of every research area.

The objective of this thesis consists of the study and development of efficient implementations for priority queues, an abstract data type employed in several applications (like CPU job scheduling and event drive simulation). Our investigation is driven by the goal of looking for high performances in the worst case.

1.2 Contents of the thesis

The thesis presents the results which have been illustrated in the following two articles:

J. Katajainen and F. Vitale, *Navigation piles with applications to sorting, priority queues and priority deques.* Nordic Journal of Computing, (10)2003, 238-262p.

C. Jensen, J. Katajainen and F. Vitale, *An experimental evaluation of local heaps,* Cph-stl report 2006-1.

In the first article we present a priority queue able to get extremely efficient performances, in terms of number of element comparisons, element moves and instructions executed for each of the required operation. In particular, with this data structure it is possible to sort a sequence of n elements with $n \log_2 n + 0.59n + O(1)$ element comparisons, $2.5n + O(1)$ element moves and $O(n \log n)$ instructions using only $4n + O(w)$ bits (where w indicates the

length of a machine word) of extra-memory (in the worst case).
It is simple to be aware of the significance of these results if one thinks that the lower bound for the number of element comparisons for sorting n elements is given by $n \log_2 n - n \log_2 e + 0.5 \log n + \Theta(1)$ [25, Section 5.3.1] while sorting algorithms which require a number of element comparisons asymptotically optimal need a superlinear amount of element movements (or bits of extra-memory when this is not even proportional to the dimension of the elements). Moreover, these algorithms are not often based on a priority queue and thus are not bound to a structure that support other operations in an efficient way.

As explained in the article, the main concepts of the navigation piles can also be applied to the construction of priority deques. Even if the experimentation of these structures is not part of the work of this thesis, it is at last important to note that the applicative potential of the navigation piles have been recently highlighted in [22], getting interesting results under suitable restrictions and modifications.

The second paper analyses, both from a theoretical and experimental point of view, a typology of heaps which support in-place operations. The key aspect which has been considered in the design and implementation of this data structure concerns an efficient cache use, a factor which is often neglected in the project of most of the algorithms. The experimental results show as local heaps, through a careful implementation which takes advantage of the localization of the elements to be accessed, lead to competitive performances with respect to the traditional heap implementations.

1.3 From heaps to navigation piles

In this section we will describe a data structure that we called heaps with extra bytes, which was born from the collaboration between Prof. Jyrki Katajainen and Fabio Vitale and it played a fundamental role in the navigation piles ideation (the data structure described in Chapter 3). In this sense this priority queue bridges the gap between the traditional heaps [38] (see also [9]) and the navigation piles.

In order improve the performances of the standard heaps for what concerns the number of element comparisons and element moves for the worst case *pop*() operation ($2 \log_2 n + O(1)$ and $\log_2 n + O(1)$ respectively, where n is

the number of elements) and for sorting ($2n \log_2 n + O(n)$ and $n \log_2 n + O(n)$) many alternative implementations have been proposed in the last decades.

The Floyd bottom-up approach, presented first in [25, Vol. 3, 2nd edition, Section 5.2.3, Exercise 18] (see also [13]), and the Carlsson [5] heapsort, have reduced the number of element comparisons for the worst case to $1.5n \log_2 n + O(1)$ [37] and $n \log_2 n + n \log_2 \log n + O(1)$ respectively.

In order to reduce to a greater extent the number of element comparisons for sort n elements, some heap variants using extra information have been proposed. MDR heaps [28] store in a separate bit sequence two bits for each non-leaf element. These two bits encode the information about which of the children has the highest value (there are three possibilities: the left or the right child has a value not smaller than the sibling value or it is not known which one is the biggest). MDR heapsort, as other sorting applications based on heap variants employing extra bits [7], requires [36] only $n \log_2 n + O(1)$ element comparisons and element moves in the worst case.

Another problem analysed and attacked in the last years concerns the improvements of the cache use (for a cache model and concepts related to it we refer to [19]). As might be expected, the fact that each element (with depth not too small) and its children belong generally to different blocks determines a high number of cache misses for the traversing of the heap. A natural solution could be the increasing of the arity of the heap, which would reduce its height [23] [26]. This way even the worst case number of element moves required by the various operations would be decreased consequently (reduced by a factor $\log_2 d$ compared to the binary case, if d is the arity of the heap). The main problem connected to this approach is the rapid increasing of the number of element comparisons (if d is bigger than 3) as a function of d.

If the key for have an asymptotically optimal number of comparisons for *pop*() and sorting using a heap priority queue seems to be related to the employing of extra memory, it should also be noted that the access to an extra-information can be as expensive as the access to the heap elements; in fact, according to the hierarchical model of the memory, the cache use will be contested by the access to the elements and the extra information.

The ***heaps with extra bytes*** allow to reach an optimal number of element comparisons for *pop*() and sorting operations with a linear amount of extra memory. At the same time this kind of heaps is able to localize the elements in order to improve the use of the cache. The data structure is con-

stituted by a d-ary heap A (where d is a power of 2) and by an array of extra information B, such that there is a $1-1$ correspondence between the set of all groups of element siblings in A and the extra information present in B. More specifically, if $A[i]$ is a non-leaf element, $B[i]$ is the extra information associated to the group of children of $A[i]$.

Each extra information $B[i]$ must be able to:

1. Allow to quickly reach (in constant time) an element whose priority is the highest among the children of $A[i]$, i.e. the top element among the group of siblings associated to $B[i]$.
2. Allow to be easily updated (with at most $O(\log d)$ instructions and $\log_2 d$ comparisons) in the case in which
   - the top element among the children of $A[i]$ is replaced by a new element.
   - the rightmost child of $A[i]$ is removed.
   - i is the index of a non-leaf element having less than d children and a new element is added in the heap.

There are several ways for implementing such kind of extra information array. We do not enter in all the details in all these possible implementations, since our goal is only to give an overview to the reader of the different possibilities to build the array of extra information.

Indirect complete sorting

Each extra information $B[i]$ stores the complete order of the children of $A[i]$. This way the first requirement can be satisfied immediately and binary search can used for the second one.

For example, if $d = 4$ we can encode the value of $B[i]$ with 5 bits storing a number between 0 and $4!-1$. This number could be updated, according to the operation executed, using a table of precomputed values able to consider all the possible cases.

If $d = 16$ each element of B could be a *long long* value (i.e. an integer of 64 bits) in which the j-th leftmost (or rightmost) subsequence of 4 consecutive bits gives us the information about the relative position of the j-th

greatest element among the associated siblings of A. The bit shift operation, assuming to be able to execute it in constant time, is very useful for satisfying the second requirement.

Indirect priority queue

By reading carefully the two above requirements it is easy to realize that an indirect complete order gives excessive amounts of information. This fact is strictly related to the fact that the extra information are a kind of indirect priority queue managing the relative group of children of a given element. For example, if $d = 32$ a solution that does not need to store the total order of all the group of element siblings could be the use of an indirect heap in which the root has only one child for each extra information (it could be implemented using 32 bytes).

The most interesting solution that we analysed is the following:
Given the value of d smaller than 512, we could encode $B[i]$ with a sequences of $d-1$ bytes storing a complete tree of offsets in level-order. In this tree the k-th leftmost leaf assumes the value 0 if $A[di + 2k-1]$ is not smaller than $A[di+2k]$, otherwise 1. We can also say that the k-th leftmost leaf points to the biggest element between $A[di + 2k-1]$ and $A[di+2k]$. Each internal node contains a value equal to the relative position of the top element among those pointed by the leaves of the subtree of which it is the root. When the top element $A[x]$ of a group of siblings is replaced by a new element, it is enough to update the $\log_2 d$ offsets on the path connecting the offset leaf pointing $A[x]$ with the root of the relative tree, executing $\log_2 d$ comparisons.

We leave to the reader the details related to the execution of these operations and to the fulfillment of the parts of the second requirement.

The use of this kind of array of extra information is simple and it will be explained according to various operations that are necessary to execute (with this description the reason of the two above requirements will appear clear):

top()
We simply return the first element of A employing only constant time.

pop()
Since we already know the top element of every group of siblings, we traverse

the heap following the path in which each element is the top child of the last element visited (employing the array B) until we reach the bottom of the heap. We then traverse bottom-up the heap looking for the correct index x of the last leaf that will be moved in one of the locations on the path we found. We finally move each element on the path connecting the top child of the root with $A[x]$ to its parent location and we move the last leaf to the $A[x]$ location. Every time that an element is substituted by another one we update the relative extra information. This operation requires $\log_2 n + O(1)$ element comparisons and $\log_d n + O(1)$ element moves in the worst case.

push(y)
We look for the correct index x of the new element employing the binary search on the path from $A[n]$ upward until the root (where n is the number of elements present in the heap before pushing the new element and $A[n-1]$ is the last element in the heap). We then move each element on that path to its top child location starting from the parent of $A[n]$ until $A[x]$ and we insert the new element. The only exception is represented by the first of these moves, since, instead of replacing the top sibling of $A[n]$ with its parent $A[(n-1)/d]$, we move $A[(n-1)/d]$ to the location of $A[n]$ and we update the relative extra information. If x is equal to n we simply insert the new element in $A[n]$, updating the relative extra information. *push*(y) does not require more than $\log_2 \log n + \log_2 d + O(1)$ element comparisons and $\log_d n + O(1)$ element moves in the worst case.

The calculation of the number of cache misses generated by the execution of the various operations is beyond the purposes of this brief description, but as might be expected it can be reduced by increasing the value of d.

We return now to the last method analysed, in which $B[i]$ is a complete tree of offsets. There are two important observations:

1. We actually do not need the root of the heap, since the first extra information stored in B already gives us the top element of the data structure.
2. It is sufficient to allocate for each offset having height h only $h+1$ bits, since the elements pointed by the leaves of the subtree of which it is

the root are at most 2^{h+1}.

Taking into account these observations, if we think of all the elements in the data structure as a unique group of siblings we would get a new idea for a data structure, which gives the main lines for the creation of the navigation piles.

1.4 Organization of the thesis

The thesis is organized in the following way. In Chapter 2 we introduce the preliminary notions which are useful for understanding the two articles. In Chapter 3 and 4 we present the contents of the first and the second article, respectively. The conclusions are present in Chapter 5.

Chapter 2

Preliminaries

In this chapter we recall some basic definitions, which are useful to understand the results we illustrate.

2.1 Algorithms and complexity

Informally an algorithm is a procedure (which always end) composed by a limited sequence of elementary operations that transforms one or more input values in one or more output values. An algorithm defines therefore an implicit function from the input set to the output set and at the same time it describes an effective procedure letting to determine for each possible input the corresponding output.

This correspondence represents therefore a solution for a given problem.

The execution of an algorithm on a given input requires the consumption of a certain resource amounts, like the time used, the memory space employed or the number of calculation devices utilized.

For our analysis the principal resources taken into consideration are the time and space used.

Usually a given problem has a natural dimension (typically identified with the size of data to elaborate) which here is indicated by the parameter n. The goal of this analysis consists of finding expressions for the amount of employed resources as function of n. We are interested in the average-case complexity $T_A^m(x)$ ($S_A^m(x)$) where x is the input. Informally this is the time (space) that we expect to be required by the algorithm. The worst case complexity $T_A^p(x)$ ($S_A^p(x)$) is instead the time (space) necessary to execute

the algorithm with the worst configuration of the input data.

Finding this function can be complicated description of these functions can be complicated, since the variable x assumes values from the set of all the inputs. A solution which can provide good information consists of the introduction of the concept of instance dimension, gathering the inputs x with the same dimension $|x|$.

Since two different instances with the same size can have different time (space) execution, we should solve the problem to define the time (space) as function of the only dimension.

A possible solution consists of considering the maximum time on all the inputs with size n; a second one can be the average time.

We call worst case complexity the function $T_A^p(n)$: $\mathbb{N} \to \mathbb{N}$ such that, for each $n \in \mathbb{N}$, $T_A^p(n) = \max\{T_A(x) : |x| = n\}$.

We call average-case complexity the function $T_A^m(n)$: $\mathbb{N} \to \mathbb{R}$ such that, for each $n \in \mathbb{N}$, $T_A^m(n) = (\sum_{|x|=n} T_A(x))/I_n$ where I_n is the number of input instances x having dimension n.

In an analogue way we can define the worst case space complexity and the average-case space complexity $S_A^p(n)$ and $S_A^m(n)$.

2.2 Asymptotic notations

The study of the asymptotic behaviour can be strongly simplified introducing some relations between numeric sequences that are currently used in this contests.

A function $f(n)$: $\mathbb{N} \to \mathbb{R}$ is "big o" of $g(n)$: $\mathbb{N} \to \mathbb{R}$, in symbols $f(n) = O(g(n))$ if there exist a real constant $c > 0$ and an integer $n_0 \geq 0$ such that, for each $n > n_0$, $f(n) \leq cg(n)$; one can say also that $f(n)$ has order of growth less or equal to that one of $g(n)$.

A function $f(n)$: $\mathbb{N} \to \mathbb{R}$ is "big omega" of $g(n)$: $\mathbb{N} \to \mathbb{R}$, in symbols $f(n) = \Omega(g(n))$ if there exist a real constant $c > 0$ and an integer $n_0 \geq 0$ such that, for each $n > n_0$, $f(n) \geq cg(n)$; one can say also that $f(n)$ has order of growth greater or equal to that one of $g(n)$.

Two functions $f(n)$: $\mathbb{N} \to \mathbb{R}$ and $g(n)$: $\mathbb{N} \to \mathbb{R}$ have the same order of growth, in symbols $f(n) = \Theta(g(n))$, if there exist two real constants c, $d > 0$ and an integer $n_0 \geq 0$, such that, for each $n > n_0$, $cg(n) \leq f(n) \leq dg(n)$.

A function $f(n)$: $\mathbb{N} \to \mathbb{R}$ is asymptotic to $g(n)$: $\mathbb{N} \to \mathbb{R}$, in symbols $f(n) \sim g(n)$, if $\lim_{n\to\infty} f(n)/g(n) = 1$.

A function $f(n)$: $\mathbb{N} \to \mathbb{R}$ is "little o" of $g(n)$: $\mathbb{N} \to \mathbb{R}$, in symbols $f(n) = o(g(n))$, if $\lim_{n\to\infty} f(n)/g(n) = 0$.

The relations O and Ω are reflexive and transitive, but not symmetric. Θ and $\sim$ are reflexive, transitive and symmetric, and so are equivalence relations.

2.3 Model of computation

The model of computation underlying our analysis is the ***RAM*** model, which is an abstraction of a computer running a program. The following description follows the main lines of [1] (for further details we refer to [18]). A RAM is provided with a read-only input device, a write-only output device, a memory consisting of a numbered sequence of cells (or registers) r_0, r_1, ..., and a program stored in a separate memory (hence we are assuming that the program can not modify itself). r_0 is called accumulator and it is the register in which take place all computation. Each cell of the memory can hold an integer of arbitrary size and the program is a sequence of instructions possibly associated with a label. Each I/O device consists of a tape holding a sequence of symbols representing integers. Whenever a symbol is read or written by the RAM, the relative tape head move one position to the right, in order to read the next symbol from the input sequence or write a new symbol on the output sequence.

Basically the program is composed by instructions able to execute arithmetic operation, branching and indirect addressing. Each instruction is composed by an own operation code and an argument. All the possible instructions can be divided in four categories:

1. Register management, concerning the loading of an integer in r_0 or the storing of the value of r_0 in a register: LOAD and STORE

2. Arithmetic operations, that are addition, subtraction, multiplication and division: ADD, SUB, MULT and DIV.

3. I/O operations: READ and WRITE.

4. branching operation: JUMP, JGTZ, JZERO.

5. halting instructions: HALT.

The instructions in the first three categories required an operand of the form, $= i$, i, or $*i$ (where i is an integer), indicating respectively the integer i itself, the integer contained in r_i and the integer contained in r_{r_i}. We assume that the machine halts whenever attempting to execute any invalid operation. The instructions of the fourth category require instead a label as argument.

Finally the instruction HALT does not require any argument.

Initially r_i is 0 for all i and, after the execution of an instruction not belonging to the fourth category, the program counter will be set to the next instruction.

The following table explains the meaning of all the instructions described.

Table 2.1: Meaning of RAM instructions.

x : operand

v_x : value of the operand x (according to the three possible forms of x)

l : label

Instructions	Effect
ADD x	$r_0 \leftarrow r_0 + v_x$
SUB x	$r_0 \leftarrow r_0 - v_x$
MULT x	$r_0 \leftarrow r_0 v_x$
DIV x	$r_0 \leftarrow \lfloor r_0 / v_x \rfloor$
LOAD x	$r_0 \leftarrow v_x$
STORE i	$r_i \leftarrow r_0$
STORE $*i$	$r_{r_i} \leftarrow r_0$
READ i	$r_i \leftarrow$ input symbol read by the tape head
READ $*i$	$r_{r_i} \leftarrow$ input symbol read by the tape head
WRITE x	v_x is written on the next cell of the output tape
JUMP l	The program counter j is set to the instruction with label l
JGTZ l	The program counter j is set to the instruction with label l if $r_0 > 0$, otherwise it is set to $j+1$
JZERO l	The program counter j is set to the instruction with label l if $r_0 = 0$, otherwise it is set to $j+1$
HALT	The execution terminates

As already explained in Section 2.1 the most important measures considered in the evaluation of the performance of a given program are the time spent and the space used during the its execution.

The concepts of worst case complexity and average case complexity for a given RAM are the simple adaptation of the definition of Section 2.1 to this contest. In particular, we need now to define the time cost of a single RAM instruction execution and the space occupied by each register.

The are two measures for the complexity of an instruction. The uniform cost criterion considers the cost of any instruction equal to one unit of time cost and the space occupied by each register equal to one unit of space. The logarithmic cost criterion considers the complexity as proportional to the number of binary digits necessary to represent all the integers used during the computation. We will use the uniform cost criterion as analysis method for the time taken by the execution of an algorithm. We refer to further details on the second criterion we refer to [1].

One of the main motivations leading to the choice of the RAM model with the uniform cost criterion for our analysis is connected to the assumption on the costs of the elementary operations. As we say in the introductory part of the navigation pile article (see Section 3.1), among the instructions that we consider available there are additions, multiplications, comparisons and operations concerning the bit manipulation (bitwise and, bitwise or, bitwise not, left shift and right shift). Saying for example that an algorithm execution requires O(1) time, we mean that it requires the execution of a constant number of some of the elementary operations that we consider available.

Chapter 3

Navigation piles

3.1 Introduction

This work is part of the CPH STL project, where the goal is to develop an enhanced edition of the Standard Template Library (STL). For more information about the project, see the CPH STL website [10].

The STL, which is an integrated part of the ISO standard for the C++ programming language [20], supports many basic data processing tasks related to sorting and searching. We describe an efficient data structure, called a navigation pile, which can be used for the realization of the sort function, the priority-queue class, and the priority-deque class, the last being a CPH STL extension of the STL. In this paper the theoretical justification of the methods proposed is given.

Our work was motivated by the experiments carried out in our project group by Jensen [21]. His results were as follows:

1. A priority queue relying on extensive pointer manipulation is doomed to fail due to the high number of cache misses.

2. Instead of binary heaps it is better to use d-ary heaps (for small values of d) since for them the memory references are more local. This confirmed the results reported earlier by LaMarca and Ladner [26, 27].

3. A priority queue relying on merging turned out to be slow when elements are large due to the high number of element moves.

When large elements are manipulated in internal memory, the most important performance indicators are the number of element comparisons and that of element moves. Therefore, we try to keep these numbers close to the absolute minimum. We still rely on (implicit) pointer manipulation, but by packing the navigation information compactly we hope that this part of the data structure is kept close to the central processing unit at all times.

As a data structure, a navigation pile is elegant and its manipulation algorithms are conceptually simple. Therefore, we feel that this structure should be presented in any modern textbook on algorithms and data structures as an addition to heaps.

The model of computation used is the ***word RAM*** as defined in [18]. We assume the availability of the following instructions: comparison, addition, multiplication, left shift, right shift, bitwise and, bitwise or, bitwise not, memory allocation, and memory deallocation functions as defined in C++. We let w denote the ***length*** of each ***machine word*** measured in bits. By an ***element move*** we mean the execution of a copying operation, a copy construction, or a copy assignment; provided for the elements being manipulated. An ***element comparison*** means the evaluation of an ordering function which returns true or false, and which defines a strict weak ordering on the set of elements. For a formal definition of a strict weak ordering, see, for example, [20, §25.3]. An ***instruction*** means any allowable word operation. Observe that the instructions executed inside the element constructor, the element destructor, the element assignment, and the ordering function are not included in our instruction counts.

For integers i and k, $i \leq k$, we use $[i\,..\,k)$ to denote the ***sequence*** of ***integers*** $\langle i, i+1, \ldots, k-1\rangle$, and $A[i\,..\,k)$ the ***sequence*** of ***elements*** $\langle A[i], A[i+1], \ldots, A[k-1]\rangle$. In the C++ standard library, it is customary to represent a sequence using a pair of iterators which indicate the position of the first element and that of the one-past-the-end element, respectively. An iterator object is assumed to provide operations so that all the elements in the sequence can be reached from the given positions. We bypass these low-level details and see a sequence as a single object.

Due to the variations in the terminology in the literature we briefly recall the concepts related to trees relevant for us. A node of a 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 root having depth 0. The ***height*** of a node is the length of the longest path from that node to a leaf. Let $d \geq 2$ be an integer. In a

complete d***-ary tree*** all its branches have d children, and all its leaves have the same depth. A ***left-complete*** d***-ary tree*** is obtained from a complete d-ary tree by removing some of its rightmost leaves.

A d***-ary heap*** with respect to an ordering function *less*() is a data structure having the following properties:

Shape: It is a left-complete d-ary tree.

Load: Each node of this tree stores one element of a given type.

Order: For each branch of the tree, the element y stored at that node is no smaller than the element x stored at any child of that node, i.e. $less(y, x)$ must return false.

Representation: The tree is represented in a sequence by storing the elements in breadth-first order.

Informally, such a data structure is often called a d-ary max-heap. The binary heaps were invented by Williams [38] and the generalization to $d > 2$ was suggested by Johnson [23].

A ***priority queue*** with respect to an ordering function *less*() is a data structure storing a set of elements and supporting the following operations:

priority_queue(**const** input_sequence& X, **const** ordering& f,
const container_sequence& Y);

Effect: Construct a priority queue containing the elements stored in the sequence referred to by X using the function referred to by f as the ordering function and a copy of the sequence referred to by Y as the underlying container for the elements. In particular, note that the elements are to be copied from the input sequence to a separate container sequence.

const element& *top*() **const**;

Effect: Return a reference to an element whose priority is highest among the elements stored in the priority queue. We call such an element the ***top element***, i.e. for that element y and for every other element x in the priority queue, $less(y, x)$ must return false.

void *push*(**const** element& x);

Effect: Insert a copy of the element referred to by x into the priority queue.

void *pop*();

Requirement: The priority queue should not be empty.

Effect: Erase the top element from the priority queue.

According to the C++ standard [20, §23.2.3], other operations (like *size*()) should be supported as well, but normally these are trivial to implement.

A ***priority deque*** (double-ended queue) is similar, but in addition we have the member function:

const element& *bottom*() **const**;

Effect: Return a reference to an element whose priority is lowest among the elements stored in the priority deque. We call such an element the ***bottom element***.

And instead of the *pop*() function we have the member functions:

void *pop_top*();

Requirement: The priority deque should not be empty.

Effect: Erase the top element from the priority deque.

void *pop_bottom*();

Requirement: The priority deque should not be empty.

Effect: Erase the bottom element from the priority deque.

In [24] a data structure having the same shape, load, and representation properties as a heap was used in the realization of resizable arrays. To distinguish the data structure from the heap it was called a ***pile***. The data structure proposed by us is a modification of a pile, where the branches store navigation information and the leaves store the elements. Therefore, we call the data structure a ***navigation pile***. One could also see the navigation pile as an extension of the selection tree discussed, for example, in [25].

Let N be a fixed power of 2. A navigation pile is a priority queue that can store at most N elements. Let n denote the number of elements in the data structure. The basic features of a navigation pile can be listed as follows:

1. The data structure requires $2N$ bits in addition to the elements stored. Using standard packing techniques, it is possible to pack the extra bits into $\lceil 2N/w \rceil$ words.

2. The construction of the data structure requires $n-1$ element comparisons, n element moves, and $O(n)$ instructions.

3. The *top*() function takes $O(1)$ instructions.

4. If the data structure is not full and it is possible to execute the *push*() function, its execution requires at most $\log_2 \log_2 n + O(1)$ element comparisons, one element move, and $O(\log_2 n)$ instructions.

5. The *pop*() function requires at most $\lceil \log_2 n \rceil$ element comparisons, two element moves, and $O(\log_2 n)$ instructions.

The data structure is described and analysed in Section 3.2, and it is made dynamic in Section 3.4 with not much loss in efficiency.

A navigation pile can be immediately used for sorting. The resulting sorting algorithm, called pilesort, sorts a sequence of n elements using at most $4n+O(w)$ extra bits of memory, $n \log_2 n + 0.59n + O(1)$ element comparisons, $2.5n + O(1)$ element moves, and $O(n \log_2 n)$ instructions. Recall that the information-theoretic lower bound for the number of element comparisons is $n \log_2 - n \log_2 e + (1/2) \log_2 n + \Theta(1)$ [25, §5.3.1] and the optimum for the number of element moves $n - T + C$, where T and C denote the number of trivial and nontrivial cycles in the permutation of elements being sorted [30]. Pilesort is described and analysed in Section 3.3.

In Section 3.5 we show how navigation piles can be employed in the implementation of priority deques. In Section 3.6 we consider some generalizations of navigation piles. In Section 3.7 the results proved are summarized (see Table 3.1) and some open problems are posed.

Before going into the details we give a brief survey of some related work. A data structure similar to our navigation pile has earlier been described by Pagter and Rauhe [32], and used for sorting. The usage in connection with a priority queue and a priority deque seems to be new. As compared to their structure we use more bits to save a significant number of element comparisons. Moreover, they allowed holes in their structure, whereas we maintain the elements compactly as done in heaps. This makes the dynamization of the structure easy. Also, the fine-heap of Carlsson et al. [7] uses similar ideas as our navigation pile.

Traditional implementations of a priority queue, like that provided in the Silicon Graphics Inc. implementation of the STL [34], are based on a binary heap, which is an ***in-place*** data structure requiring only $O(1)$ extra words of memory. As a navigation pile, a binary heap is ***static*** in a sense that the maximum number of elements to be stored must be known beforehand. Using the techniques described, for example, in [24] — also used by us — the structure can be dynamized space-efficiently and bounds similar to ours are obtained, except that in the worst case the *push*() and *pop*() functions require a logarithmic number of element moves. It is well-known that the efficiency of binary heaps can be improved by storing extra bits at the nodes (see, for example, [7, 12, 36]). Similarly, for small values of d, the efficiency of d-ary heaps can be improved in a space-efficient manner to use the same number of element comparisons as improved binary heaps, but for the *push*() and *pop*() functions the number of element moves performed in the worst case gets reduced to $\log_d n + O(1)$.

As to sorting, there are variants of heapsort (see, e.g. [7, 12, 36]) that use n extra bits and require $n \log_2 n - \Omega(n)$ or $n \log_2 n + O(n)$ element comparisons, but the number of element moves can be as high as that of element comparisons. Independently of our work, Franceschini and Geffert [14] have devised an in-place sorting algorithm that performs $2n \log_2 n + o(n \log_2 n)$ element comparisons and less than $13n + \varepsilon n$ element moves, where ε is arbitrarily small, but fixed, real number greater than zero. Moreover, there exists an in-place sorting algorithm [30] which performs the optimum number of element moves, but then the amount of other operations gets high.

Three static in-place priority-deque structures have been proposed in the literature: min-max heaps [2], deaps [4] (see also [8]), and interval heaps [35]. All these structures can be made dynamic using the techniques described in [24], after which the efficiency of the functions *push*(), *pop_top*(), and *pop_bottom*() would be about the same as for our structure, except that in the worst case the number of element moves is logarithmic, whereas we need only a constant number of element moves per operation.

3.2 Navigation piles

Let N be a fixed power of 2, i.e. $N = 2^\eta$ for some nonnegative integer η. A ***navigation pile*** is a priority queue having the following properties:

Shape: It is a complete binary tree of size $2^{\eta+1} - 1$.

Leaves: If there are n elements, $n \leq 2^\eta$, the first n leaves store one element each and the remaining leaves are empty.

Branches: We say that the leaves in the subtree rooted by a branch form the ***leaf sequence*** of that branch. Each branch, whose leaf sequence contains elements, stores the index of the leaf inside the leaf sequence containing the top element among all the elements stored in the leaf sequence. As earlier the top element is defined with respect to a given ordering function *less*().

Representation: Since the types of data stored at the leaves and branches are different, the data structure consists of two sequences: $A[0\,..\,n)$ storing the elements and $B[0\,..\,2^{\eta+1})$ storing the navigation information.

The data structure is illustrated in Fig. 4.2 for integer data using **operator<**() as the ordering function.

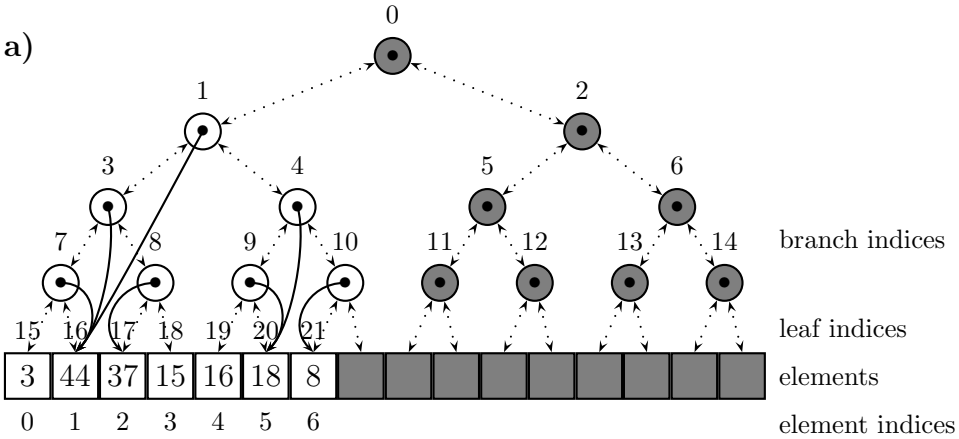
We use pseudo C++ in the description of our algorithms. In particular, we omit all low-level details related to bit manipulation and memory allocation. For the sake of clarity, we divide the integers operated on into several overlapping categories: ***levels*** are nonnegative integers no larger than η, ***offsets*** are the indices stored at the branches, ***branch indices*** are the indices of the branches of the complete binary tree, ***leaf indices*** are the indices of the leaves of that tree, ***element indices*** are the indices of the elements stored in $A[0\,..\,n)$, and ***bit indices*** are the indices of the bits stored in $B[0\,..\,2^{\eta+1})$. The branch and leaf indices are collectively called ***node indices***. For instance, the element stored at the leaf with leaf index i has the element index $i-2^\eta+1$ in sequence $A[0\,..\,n)$.

Let us first verify that all the offsets can be actually stored in a sequence of $2^{\eta+1}$ bits. First, there are 2^δ branches whose depth is δ, $0 \leq \delta < \eta$. Second, the leaf sequence of a branch whose height is γ stores at most 2^γ elements. Hence, γ bits are enough to represent the corresponding offset. Letting $\gamma = \eta-\delta$, we get that the total number of bits used for the offsets is

$$\sum_{\delta=0}^{\eta-1} 2^\delta(\eta-\delta) < 2^{\eta+1} = 2N\,.$$

We want to store the offsets maintained in the branches as compactly as possible. For this purpose, we assume that we have a class which stores the $2^{\eta+1}$ bits in $2^{\eta+1}/w$ words and supports the operations:

a)



b)

branch index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
offset (binary)	****	001	***	01	01	**	**	1	0	1	0	*	*	*	*

Figure 3.1: **a)** A navigation pile of size 7 and capacity 16. The normal parent/child relationships are shown with dotted arrows and the references indicated by the offsets with solid arrows. The gray nodes are not in use. **b)** The bit representation of the navigation information. Bits marked with * are not in use.

offset *get*(bit_index i, level γ) **const**;

Requirement: $\gamma \leq w$.

Effect: Return the integer corresponding to the bit sequence $B[i\,..\,i{+}\gamma)$. We use the shorthand notation $\lambda \leftarrow B[i\,..\,i{+}\gamma)$ for this operation.

void *set*(bit_index i, level γ, offset λ);

Requirement: $\gamma \leq w$.

Effect: Update the contents of the bit sequence $B[i\,..\,i{+}\gamma)$ using the γ low-order bits of λ. In brief, we denote this operation as $B[i\,..\,i{+}\gamma) \leftarrow \lambda$.

The implementation of these functions is straightforward using $O(1)$ left and right shifts, and bitwise boolean instructions.

The navigation-pile class has several member functions which simplify the description of the other member functions. To make the bounds for the number of element comparisons and machine instructions depend on n, not on N, any of the nodes on the left arm of the complete binary tree can be the root. Below we give the first collection of member functions.

level *depth*(node_index i) **const**;

Effect: return $\lfloor \log_2(1+i) \rfloor$;

level *height*(node_index i) **const**;

Effect: return $\eta - depth(i)$;

node_index *first_child*(node_index i) **const**;

Effect: return $2i+1$;

node_index *second_child*(node_index i) **const**;

Effect: return $2i+2$;

node_index *parent*(node_index i) **const**;

Effect: return $\lfloor (i-1)/2 \rfloor$;

node_index *ancestor*(node_index i, level Δ) **const**;

Requirement: $\Delta \leq depth(i)$.

Effect: return $\left\lfloor (i+1)/2^{\Delta} - 1 \right\rfloor$;

node_index *root*() **const**;

Effect: level $\delta \leftarrow \eta - \lceil \log_2 n \rceil$;
return $2^{\delta} - 1$;

leaf_index *first_leaf*() **const**;

Effect: return $2^{\eta} - 1$;

leaf_index *last_leaf*() **const**;

Effect: **return** $2^\eta + n - 2$;

element_index *size*() **const**;

Effect: **return** n;

bool *is_first_child*(node_index i) **const**;

Effect: **return** (i **bitand** $1 = 1$);

bool *is_root*(node_index i) **const**;

Effect: **return** ($i = root()$);

bool *is_in_use*(node_index i) **const**;

Effect: level $\gamma \leftarrow height(i)$;
leaf_index *start_of_leaf_sequence* $\leftarrow 2^\gamma i + 2^\gamma - 1$;
return ($\gamma \leq \lceil \log_2 n \rceil$ **and** *start_of_leaf_sequence* $\leq$ *last_leaf*());

bit_index *start_of_offset*(branch_index i) **const**;

Effect: level $\delta \leftarrow depth(i)$;
level $\gamma \leftarrow height(i)$;
branch_index *index_at_own_level* $\leftarrow i - 2^\delta + 1$;
bit_index *bits_upto_this_level* $\leftarrow (\eta+1)\delta(\delta+1)/2 - \delta(\delta+1)(2\delta+1)/6$;
bit_index *bits_before* $\leftarrow$ *bits_upto_this_level* + *index_at_own_level* $\cdot \gamma$;
return *bits_before*;

leaf_index *element_to_leaf*(element_index ℓ) **const**;

Effect: **return** *first_leaf*() $+ \ell$;

element_index *leaf_to_element*(leaf_index L) **const**;

Effect: **return** $L -$ *first_leaf*();

(element_index, offset) *jump_to_element*(branch_index i) **const**;

Effect: bit_index $s \leftarrow$ *start_of_offset*(i);
level $\gamma \leftarrow$ *height*(i);
offset $\lambda \leftarrow B[s\,..\,s{+}\gamma)$;
leaf_index *start_of_leaf_sequence* $\leftarrow 2^\gamma i + 2^\gamma - 1$;
element_index $\ell \leftarrow$ *leaf_to_element*(*start_of_leaf_sequence*$+\lambda$);
return (ℓ, λ);

All these member functions are quite straightforward, except the function *start_of_offset*(). It is supposed to calculate the number of bits used by the offsets stored prior to the offset of the given branch, when the offsets of all branches are stored in breath-first order. Let δ and γ be the depth and height of the given branch, respectively. The desired number can be obtained by summing the number of bits used by the offsets of the branches on the full levels above, which is $\sum_{\beta=0}^{\delta-1} 2^\beta(\eta - \beta)$ bits in total, and the number of bits used by the offsets stored at the nodes on the same level, which is γ bits per branch. The formula used in the pseudo code is the above sum in a closed form.

The correctness of these member functions follows directly from the properties of navigation piles. Assuming that the whole-number logarithm function is also in our repertoire of constant-time operations, all the functions clearly execute at most $O(1)$ instructions. At the end of this section we explain how to abandon the constant-time logarithm function without increasing the resource bounds for the public member functions: constructor, *top*(), *push*(), and *pop*().

Now we are ready to present the realization of the priority queue operations. We start with the constructor. The basic idea is simple: visit the branches that are in use in a bottom-up manner and for each branch update the offset using the information available at its children. We follow the recommendation of Bojesen et al. [3] and visit the nodes in depth-first order to improve the cache performance. Instead of relying on recursion, our implementation is iterative.

navigation_pile(**const** input_sequence& X, element_index *capacity*,
const ordering& f, **const** container_sequence& Y);

Effect: Allocate space for the sequence $A[0\,..\,capacity)$ of the same type as the sequence referred to by Y;
Copy the elements stored in the sequence referred to by X to A;
Construct a private copy of the ordering function referred to by f and call it $less()$;
Let 2^n be the smallest power of 2 larger than or equal to $capacity$;
Allocate space for the bit sequence $B[0\,..\,2^{n+1})$;
Use n as a synonym for $A.size()$;
if $(n \leq 1)$ **return**;
$make\_pile()$;

The $make\_pile()$ function and its utility functions are as follows.

void $make\_pile()$;

Effect: branch_index $j \leftarrow parent(first\_leaf())$;
branch_index $\ell \leftarrow parent(last\_leaf())$;
for (branch_index $k \leftarrow \ell$; $k \geq j$; $--k$)
  $handle\_height\_one\_branch(k)$;
  branch_index $i \leftarrow k$;
  while $(is\_first\_child(i)$ **and not** $is\_root(i))$
    $i \leftarrow parent(i)$;
    $handle\_upper\_branch(i)$;

void $handle\_height\_one\_branch$(branch_index i);

Effect: bit_index $s \leftarrow start\_of\_offset(i)$;
element_index $\ell \leftarrow leaf\_to\_element(first\_child(i))$;
if $(is\_in\_use(second\_child(i)))$
  element_index $m \leftarrow leaf\_to\_element(second\_child(i))$;
  $B[s\,..\,s{+}1) \leftarrow less(A[m], A[\ell])\ ?\ 0\ :\ 1$;
else
  $B[s\,..\,s{+}1) \leftarrow 0$;

void $handle\_upper\_branch$(branch_index i);

Effect: bit_index $s \leftarrow$ *start_of_offset*(i);
level $\gamma \leftarrow$ *height*(i);
(element_index ℓ, offset λ) $\leftarrow$ *jump_to_element*(*first_child*(i));
if (*is_in_use*(*second_child*(i)))
(element_index m, offset μ)$\leftarrow$
jump_to_element(*second_child*(i));
$B[s\,..\,s{+}\gamma) \leftarrow$ *less*$(A[m], A[\ell])$? λ : $2^{\gamma-1}{+}\mu$;
else
$B[s\,..\,s{+}\gamma) \leftarrow \lambda$;

For each branch with both children in use one element comparison is performed. The number of such branches is $n{-}1$, which gives us the number of element comparisons performed. At the beginning the elements are copied to the container sequence, but thereafter the elements are not moved. That is, exactly n element moves are performed. Most code is needed for performing transformations between different kinds of indices, but all these take only $O(1)$ instructions. Since each branch in use is visited once, the total number of instructions executed is $O(n)$.

The *top*() function is easily realized by following the offset stored at the root. Clearly, only $O(1)$ instructions are needed.

const element& *top*() **const**;

Effect: (element_index ℓ,$\cdot$) $\leftarrow$ *jump_to_element*(*root*());
return $A[\ell]$;

If $n < N$ and the space originally allocated for the container sequence is not exhausted, the *push*() function can be accomplished by appending the new element at the end of sequence $A[0\,..\,n)$ (we assume that this is done by the *push_back*() function), and updating the navigation information at the branches. A naive way to do the updates is to visit the branches on the ***special path*** from the last leaf to the root one by one starting from the bottom. In practice this method may be sufficient, but in the worst case it requires $\lceil \log_2 n \rceil$ element comparisons. As for heaps (see, for example, [16]), binary search can be used to accelerate the location of the branch whose offset refers to an element that is smaller than the new element and whose height is the largest for such branches; if no such branch exists, the last leaf is output. When the index of this branch is available, the offsets of the branches on the special path up to this branch are updated to refer to the last leaf. The implementation details are given below.

void *push*(**const** element& x);

Effect: A.*push_back*(x);
node_index $i \leftarrow$ *binary_search_on_special_path*(x);
update_partial_path(*last_leaf*(), i, *last_leaf*());

node_index *binary_search_on_special_path*(**const** element& x);

Effect: level $\Delta \leftarrow$ *height*(*root*());
node_index $j \leftarrow$ *last_leaf*();
while ($\Delta > 0$)
  level *half* $\leftarrow \lfloor \Delta/2 \rfloor$;
  branch_index $i \leftarrow$ *ancestor*($j, \Delta - half$);
  (element_index ℓ, $\cdot$) $\leftarrow$ *jump_to_element*(i);
  if (*less*($A[\ell], x$))
    $j \leftarrow i$;
    $\Delta \leftarrow$ *half*;
  else
    $\Delta \leftarrow \Delta - half - 1$;
return j;

void *update_partial_path*(node_index j, node_index i, leaf_index K);

Requirement: j is an ancestor of K and i is an ancestor of j.

Effect: level $\gamma \leftarrow$ *height*(j);
while ($j \neq i$)
  $j \leftarrow$ *parent*(j);
  $\gamma \leftarrow \gamma + 1$;
  bit_index $s \leftarrow$ *start_of_offset*(j);
  leaf_index *start_of_leaf_sequence* $\leftarrow 2^\gamma j + 2^\gamma - 1$;
  offset $\lambda \leftarrow K - start\_of\_leaf\_sequence$;
  $B[s \,..\, s+\gamma) \leftarrow \lambda$;

In the *push*() function only binary search involves element comparisons. Since the length of the path considered is $\lceil \log_2 n \rceil$, $\lceil \log_2(\lceil \log_2 n \rceil + 1) \rceil$ element comparisons are done. One element move is needed since the parameter for the function is a **const** object. Clearly, the number of instructions executed is $O(\log_2 n)$.

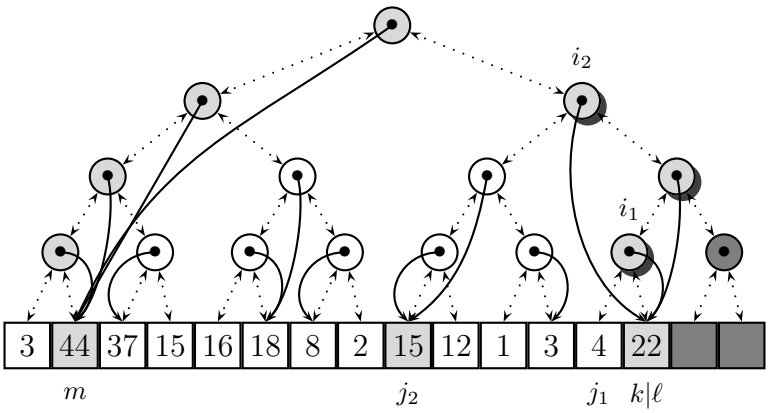


Figure 3.2: Illustration of Case 3. The nodes whose contents may change are indicated in light gray. When updating the contents of the shadowed branches on the right, no element comparisons are necessary.

In a naive implementation of the *pop*() function the top element is overwritten by the element taken from the last leaf, that element is erased (using the *pop_back*() function), and the navigation information is updated accordingly. In our implementation the top element can get destroyed in three different ways (see Fig. 3.2). Let i_1 and i_2 be the branch index of the first and the second ancestor of the last leaf having two children. If prior to the function call $n > 1$, the branch with index i_1 exists, and if $n-1$ is not a power of 2, the branch index i_2 exists. Let m be the element index of the top element, ℓ the element index of the last leaf, k the element index referred to by the offset stored at the branch with index i_2, j_1 the element index referred to by the offset stored at the first child of the branch with index i_1 (or $j_1 = \ell - 1$ if ℓ is odd), and j_2 the element index referred to by the offset stored at the first child of branch with index i_2.

Case 1: If $m = \ell$, the top element stored at the last leaf is erased, and the offsets on the path from the new last leaf to the root are updated.

Case 2: **a)** If $m \neq \ell$ and $k \neq \ell$, or **b)** if $m \neq \ell$ and ℓ is a power of 2, the assignment $A[m] \leftarrow A[\ell]$ is performed, the element copied is erased, the offsets stored at the branches on the path from i_1 (including it) up to i_2 (excluding it) are updated to refer to the leaf corresponding to the position j_1 (if they referred earlier to the last leaf), and the offsets on

the path from the leaf corresponding to the position m up to the root are updated.

Case 3: If $m \neq \ell$ and $k = \ell$, the assignments $A[m] \leftarrow A[j_2]$ and $A[j_2] \leftarrow A[\ell]$ are done, the element stored at the last leaf is erased, the offsets stored at the branches on the path from i_1 (including it) up to i_2 (excluding it) are updated to refer to the leaf corresponding to the position j_1, the offsets on the path from i_2 (including it) upwards are updated to refer to the leaf corresponding to the position j_2 if they referred earlier to the last leaf, and the offsets on the path from the leaf corresponding to the position m up to the root are updated.

A more detailed description of the *pop*() function and its utility functions is given below.

void *pop*();

Effect:
```
if (n = 1)
    A.pop_back();
    return;
(element_index m, ·) ← jump_to_element(root());
const element_index ℓ ← leaf_to_element(last_leaf());
(branch_index i1, branch_index i2) ←
    first_two_ancestors_with_more_than_one_child(last_leaf());
(element_index k, ·) ← jump_to_element(i2);
element_index j1;
if (height(i1) = 1)
    j1 ← ℓ − 1;
else
    (j1, ·) ← jump_to_element(first_child(i1));
if (m = ℓ)
    A.pop_back();
    update_full_path(last_leaf());
elseif (k ≠ ℓ)
    A[m] ← A[ℓ];
    A.pop_back();
    update_partial_path(i1, i2, element_to_leaf(j1), last_leaf());
```

update_full_path(*element_to_leaf*(m));
else
(element_index j_2, $\cdot$) $\leftarrow$ *jump_to_element*(*first_child*(i_2));
$A[m] \leftarrow A[j_2]$;
$A[j_2] \leftarrow A[\ell]$;
A.*pop_back*();
update_partial_path(i_1, i_2, *element_to_leaf*(j_1), *last_leaf*());
update_partial_path(i_2, *root*(), *element_to_leaf*(j_2), *last_leaf*());
update_full_path(*element_to_leaf*(m));

(branch_index, branch_index)
first_two_ancestors_with_more_than_one_child(leaf_index L) **const**;

Requirement: $n > 1$.

Effect: branch_index $i_1 \leftarrow$ *parent*(L);
while (**not** *is_in_use*(*second_child*(i_1)))
$i_1 \leftarrow$ *parent*(i_1);
if (*is_root*(i_1))
return (i_1, i_1);
branch_index $i_2 \leftarrow$ *parent*(i_1);
while (**not** *is_in_use*(*second_child*(i_2)))
$i_2 \leftarrow$ *parent*(i_2);
return (i_1, i_2);

void *update_full_path*(leaf_index L);

Requirement: $n > 1$.

Effect: branch_index $i \leftarrow$ *parent*(L);
handle_height_one_branch(i);
while (**not** *is_root*(i))
$i \leftarrow$ *parent*(i);
handle_upper_branch(i);

void *update_partial_path*(branch_index j, branch_index i, leaf_index K, leaf_index L);

Requirement: i is an ancestor of j, K and L are in the leaf sequence of j.

Effect: **while** $(i \neq j)$
bit_index $s \leftarrow start\_of\_offset(j)$;
level $\gamma \leftarrow height(j)$;
offset $\kappa \leftarrow B[s\,..\,s{+}\gamma)$;
leaf_index $start\_of\_leaf\_sequence \leftarrow 2^\gamma j{+}2^\gamma{-}1$;
offset $\lambda \leftarrow L{-}start\_of\_leaf\_sequence$;
if $(\kappa \neq \lambda)$ **return**;
offset $\mu \leftarrow K{-}start\_of\_leaf\_sequence$;
$B[s\,..\,s{+}\gamma) \leftarrow \mu$;
$j \leftarrow parent(j)$;

In each of the three cases only one path update involves element comparisons. Since the depth of the root is $\lceil \log_2(n{-}1) \rceil$ after the element removal and at most one element comparison is done at each level, the number of element comparisons performed is bounded by $\lceil \log_2(n{-}1) \rceil$. No move, one move, or two moves are done depending on the case. The bound $O(\log_2 n)$ for the number of instructions is obvious.

The whole-number logarithm function has been used for two purposes: to compute the height of the tree (in the functions *root*() and *is_in_use*()) and to compute the depth or the height of a node (in the functions *depth*() and *height*()). The first usage can be avoided by remembering the height and the index of the root. The second usage can be avoided by forwarding the height of the node being manipulated in one of the parameters to the member functions that use *depth*() or *height*(). When the height is known, the depth is obtained using η. To sum up, the whole-number logarithm function can be avoided altogether.

3.3 Sorting

The *sort*() function has the following abstract interface:
void *sort*(sequence& X, **const** ordering& f);
The task is to reorder the elements stored in the sequence referred to by X such that the ordering function referred to by f returns false for the reverse of all consecutive pairs of elements.

A navigation pile can be used for sorting in the same way as a heap in heapsort [38]. We make only two minor modifications to the functions described in the previous section:

1. In the construction of the data structure the copying of the elements to a separate sequence is not necessary, but the input sequence can be used for the same purpose. To enable this a new constructor is provided which takes a handle to a sequence, not a **const** sequence, as its first parameter. We assume that such a construction transfers the ownership of the data to the navigation pile. To get the sorted output back to the initial sequence, conversion function **operator** *sequence*() is provided which converts a navigation pile to a sequence by discarding the navigation information. The manipulation of the handles to these data structures is assumed to take $O(1)$ instructions.

2. In connection with the *pop*() function the top element is not overwritten but saved at the earlier last leaf. (Actually, the C++ standard requires that the *pop_heap*() function, which is not discussed here, should just have this effect [20, §25.3.6].) To accomplish this, no change, a swap, or a rotation of three elements is performed depending on the case we are in. This new function is called *pop_and_save*().

The basic version of ***pilesort***, as it is called here, works as follows:
void *pilesort*(sequence& X, **const** ordering& f);

Effect: navigation_pile $P(X, X.size(), f)$;
while $(P.size() > 1)$
$P.pop\_and\_save()$;
$X \leftarrow P.$**operator** *sequence*();

When sorting n elements, the amount of extra space needed is at most $4n$ bits, since $2^{\lceil \log_2 n \rceil} < 2n$. Otherwise, only a constant number of additional words is used. During the construction $n-1$ element comparisons are performed. The consecutive invocations of the *pop_and_save*() function incur at most $\sum_{k=2}^{n} \lceil \log_2(k-1) \rceil$ element comparisons. Since the sum $\sum_{k=2}^{n} \lfloor \log_2 k \rfloor$ is less than $n \log_2 n - 1.91n$ (see, for example, [36]), the total number of element comparisons is bounded by $n \log_2 n + 0.09n + O(1)$. A swap requires three moves and a rotation of three elements four moves, so the total number of element moves is never more than $4n$. Since the construction of the pile requires $O(n)$ instructions, its deconstruction $O(n)$ instructions, and each invocation of the *pop_and_save*() function $O(\log_2 n)$ instructions, the total number of instructions executed is $O(n \log_2 n)$.

The number of element moves is actually at most $3.5n+O(1)$, which is seen as follows. For every odd value of $\ell = n-1$, if we have to do three moves to erase the last leaf meaning that the offset stored at the second ancestor (cf. the explanation in connection with the *pop*() function) having two children refers to the last leaf, then in the next iteration the first ancestor of the last leaf is the previous second ancestor so it cannot refer to the last leaf, or ℓ is a power of 2, and in both of these cases at most two moves are necessary.

The number of element moves can be reduced to $2.5n+O(1)$ by finding the bottom element, keeping a separate copy of it, and moving the element stored at the last leaf to the place of the bottom element, which creates a hole at the end of the sequence. In the *pop_and_save*() function a rotation of three elements can be replaced with three moves: the top element is moved to the hole, the middle element to the place of the top element, and the last of the remaining elements to the place of the middle element creating a new hole. Similarly, a swap can be replaced with two moves. After all *pop_and_save*() operations the bottom element is moved to the hole at the beginning of the sequence. Due to the search of the bottom element the number of element comparisons is increased by $n-1$. This number could even be reduced to $\lfloor (n-1)/2 \rfloor$ by finding the bottom element during the construction of the navigation pile. Just after the navigation information is available at the branches of height one, the losers are used to find the bottom element and the element stored at the last leaf is moved to its place, after which the construction of the navigation pile is continued. That is, the number of element comparisons becomes $n \log_2 n+0.59n+O(1)$.

3.4 Priority queues

In Section 3.2 we assumed that the maximum number of elements to be stored in the data structure is known beforehand. In this section we devise a fully dynamic priority queue, which consists of a collection of navigation piles instead of only one. Our construction relies heavily on the dynamization techniques developed by Katajainen and Mortensen [24].

Let us first consider the dynamization of the container sequence storing the elements. Any ***resizable array***, which is a data structure supporting the grow and shrink operations at the back end — i.e. the functions *push_back*() and *pop_back*() used in Section 3.2 — could be used for this purpose. To be

sure that the number of element moves stated in our resource bounds stay valid, we recommend that one of the two resizable-array structures described in [24] is used. The first structure based on doubling allocates space for at most $4n$ elements (this can be easily improved to $2n+O(1)$) and $O(\log_2 n)$ pointers if the sequence stores n elements. The second space-efficient structure reserves never space for more than $O(\sqrt{n})$ extra elements and $O(\sqrt{n})$ pointers. It is significant, and this is true for both of these structures, that the elements are not moved because of the dynamization after they have been inserted into the structure.

For the dynamization of the bit sequence we cannot use the space-efficient resizable arrays, but the alternative relying on doubling is still applicable. We divide the element sequence into ***blocks*** B_* and B_h of size 2^5 and 2^h for $h = 5, 6, \ldots$, respectively, until all the elements can be stored. The space for block B_* and the corresponding bit sequence is statically allocated to avoid repeated memory allocations and deallocations for small blocks. We allow the largest block to be empty for a while to deamortize the costs of memory allocations and deallocations done in the proximity of block boundaries. If there is an empty block and the largest nonempty block lacks more than 2^5 elements, the bit sequence allocated for the largest block is freed. The size of the bit sequence for each block is fixed, so for each of them the techniques discussed in Section 3.2 can be used.

The data structure is illustrated in Fig. 3.3. There are four types of headers: the ***bit header*** stores the pointers to the bit sequences, the ***segment header*** stores the start addresses of the arrays storing pointers to the memory segments allocated for the elements, the ***iterator header*** stores iterators to the top elements of the respective blocks, and the ***top header*** stores cursors to the iterator headers and gives this way the sorted order of the top elements. The latter two headers are used to facilitate a fast *top*() function. For the maintenance of the segment header and the arrays pointed to, we refer to [24]. Each header can be realized using a std::vector. Since in each header the number of entries in use is $O(\log_2 n)$, the manipulation of the headers does not destroy our bounds for the number of instructions.

The navigation information stored for B_h is a sequence of 2^{h+1} bits. In the worst case the largest block is empty and the first nonempty block lacks exactly 2^5 elements. Hence, if there are n elements in the structure, we use at most $4n+O(1)$ bits for all bit sequences.

Let us next consider the implementation of the priority queue operations one at a time. For a sequence of n elements, the construction of navigation

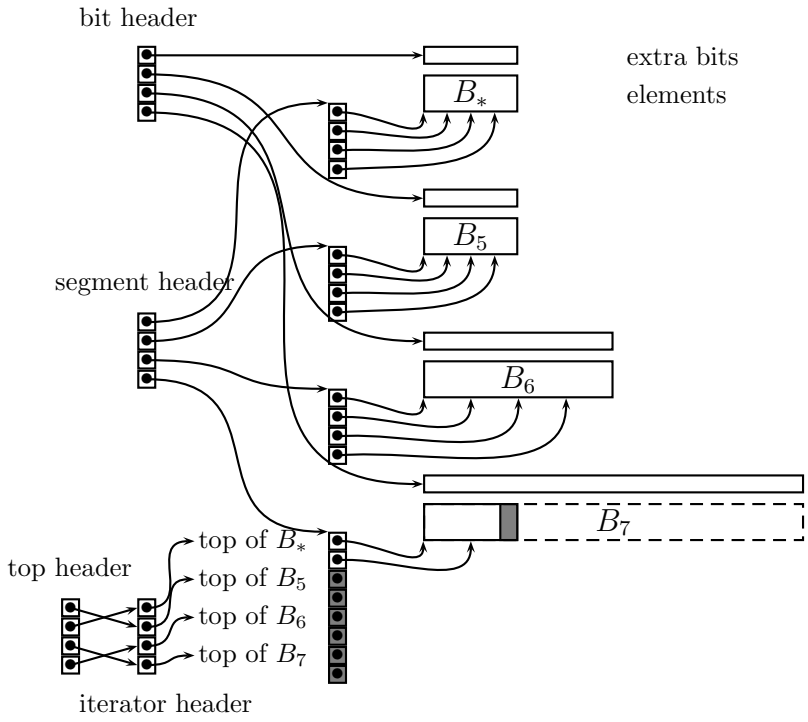


Figure 3.3: A collection of navigation piles storing 77 elements. A space-efficient resizable array is used as the container for the elements. The gray areas denote empty memory segments allocated for dynamization purposes.

piles needed can be easily carried out blockwise. Of the at most $1+\lfloor \log_2 n \rfloor$ nonempty blocks all except the last one gets full. For a full block of size 2^h exactly 2^h-1 element comparisons are done. From this and the analysis of Section 3.2 — recalling that the top header must be sorted — it follows that $n+\Theta(\log_2 n \log_2 \log_2 n)$ element comparisons, n element moves, and $O(n)$ instructions are performed in total.

We assume that the *push*() and *pop*() functions maintain the top and iterator headers. When these are available, the *top*() function can be readily executed using $O(1)$ instructions by following the cursor to the iterator header and further the iterator to the top element.

The *push*() function inserts the given element into the largest nonempty block; if that block is full, the element is inserted into a new larger block and a navigation pile containing this single element is constructed. Since the size of the largest nonempty block is at most $n/2$, the *push*() function in that block takes at most $\log_2 \log_2 n+O(1)$ element comparisons. The construction of a navigation pile of size one requires $O(1)$ instructions. After the insertion, if the top element of the largest nonempty block changed, the iterator and top headers are updated. This requires $\log_2 \log_2 n+O(1)$ additional element comparisons. To sum up, at most $2 \log_2 \log_2 n+O(1)$ element comparisons, one element move, and $O(\log_2 n)$ instructions are carried out.

The *pop*() function erases the top element by overwriting it with an element taken from the largest nonempty block. The selection of the replacing element is based on the first and the second ancestors of the last leaf having two children. Thereafter a partial path update may be carried out in the last nonempty block and a full path update in the block that contained the top element. For the fast *top*() function, we must update the top and iterator headers. The top element of the last nonempty block does not change (unless it became empty), but that of the other block may change. Hence, one binary search may be necessary to keep the top elements of the blocks in sorted order. After the correct place of a new top element is located, $O(\log_2 n)$ moves of cursors might be necessary in the top header. To sum up, the *pop*() function performs at most $\log_2 n+\log_2 \log_2 n+O(1)$ element comparisons, two element moves, and $O(\log_2 n)$ instructions.

3.5 Priority deques

To transfer our static and dynamic priority queues to priority deques, we use the twin technique described in [25, p. 645]. We pair the elements, and order the elements in the resulting pairs. If the number of elements being stored is odd, we keep one element in its own block $B_\#$. This way the elements get partitioned into two disjoint collections: the ***top-element candidates*** and the ***bottom-element candidates***. These collections can be handled separately using our earlier techniques: in the priority queue for the top-element candidates the ordering function *less*() is used and in the priority queue for the bottom-element candidates its converse function is used.

The construction of a priority deque can be done is two phases: first the elements are compared pairwise and moved into their respective candidate collections, and then the two priority queues are constructed. For an input sequence of size n, the first separation phase requires $\lfloor n/2 \rfloor$ element comparisons and n element moves. In the second phase the elements are not moved any more. Using the bounds derived for priority queues, the total number of element comparisons is never more than $1.5n+O(1)$ in the static case and $1.5n+\Theta(\log_2 n \log_2 \log_2 n)$ in the dynamic case, and that of element moves is n in both cases. If the *top*() and *bottom*() functions are to have a sublinear complexity, the creation of these data structures cannot be sped-up much because $\lceil 3n/2 \rceil - 2$ element comparisons are necessary to find both the maximum and minimum of n elements (see, for example, [33, §3]). The space consumption of the static and dynamic priority deques cannot be larger than two times that of a corresponding priority queue storing $n/2$ elements.

The resource bounds for the *top*() and *bottom*() functions are the same as those for the *top*() function in the priority queues. We only have to remember the element in $B_\#$ if there is any. In the static case, this may cause one extra element comparison. In the dynamic case, $B_\#$ can be seen as the other blocks, so for all $n \geq 1$ the number of blocks is bounded by $2+\lfloor \log_2 n \rfloor$ in both priority queues.

The *push*() function must also take the element kept in $B_\#$ into consideration. There are two cases to consider. If $B_\#$ is empty, the element being inserted is copied there. If $B_\#$ contains an element, this and the new element become twins, and this pair is added to the data structure. An element comparison is performed to know which of the elements is inserted into the top-element candidate collection and which into the bottom-element candidate collection. In the dynamic case, the pointers to the current top

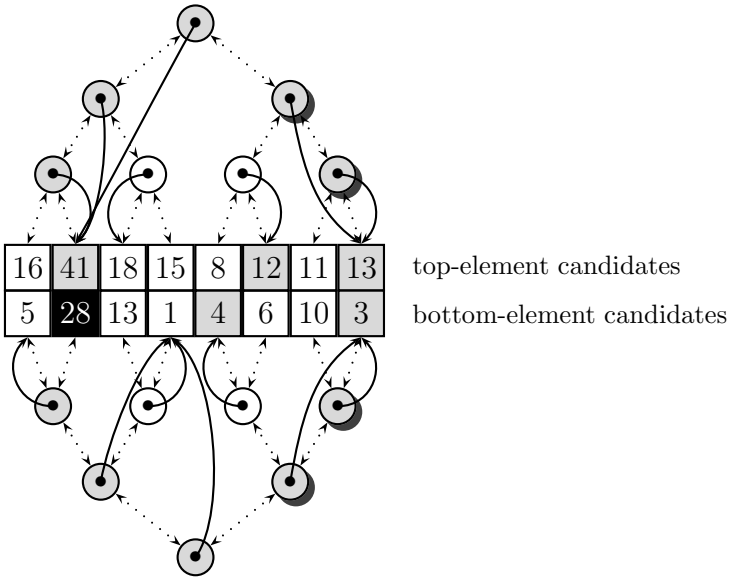


Figure 3.4: Illustration of the *pop_top*() function. We assume that the last nonempty blocks are only of size 8, that the top element is inside the last nonempty block, and that $B_\#$ is empty. The vertical alignment of the leaves indicates the twin relationship. As in Fig. 3.2, the nodes whose contents may change are indicated in light gray; for the shadowed branches the update can always be done without element comparisons. The element in the black leaf will be moved to $B_\#$.

and bottom elements are updated if necessary, which may require two binary searches. Hence, the resource bounds are double as high as the corresponding bounds for the priority queues.

The *pop_top*() and *pop_bottom*() functions are symmetric so we consider only the first one. Again there are two cases depending on the contents of $B_\#$ (for an illustration, see Fig. 3.4). If $B_\#$ is empty, we destroy the top element and move its twin to $B_\#$, take two elements from the last nonempty blocks — using the first and the second ancestors of the last leaf having two children for both blocks — and after a single element comparison move the selected elements to the holes created. Thereafter, a partial path update is performed in the last nonempty blocks, and a full path update in the blocks that received new elements. For the correctness it is important to

note that an element taken from the leaf referred to by the offset stored at the first child of the second ancestor of the last leaf having two children is replaced with a nonsmaller element taken from the last leaf, so even after the replacement the new element is not smaller than its twin. In the worst case five element moves may be necessary to get the elements involved into their correct locations. If $B_\#$ contains an element, this can be used as a twin for the twin of the destroyed element. After a single element comparison the elements are moved to their correct locations and the *update_full_path*() function is invocated at most twice. In the dynamic case, at the end the top and iterator headers are updated if necessary using binary search. Again the resource bounds can be about twice as high as those for the priority queues.

3.6 Generalizations of navigation piles

In this section we briefly discuss some generalizations of navigation piles. Our goal is to find ways to reduce the height of the pile and hereby reduce the time needed for index manipulations.

Pagter and Rauhe [32] proposed a variant where a bunch of elements forming a ***bucket*** is stored at each leaf. A further idea, which is crucial in their construction, is to partition the buckets hierarchically into blocks whose size is a power of 2, and extend the branches to contain a pointer to the block containing the top element. For a branch having height γ, the block pointer contains $\min\{\gamma, \log_2 B\}$ bits, where B — a power of 2 — is the bucket size. That is, the number of navigation bits at each branch is at most doubled, but the number of bits needed for the whole tree is proportional to n/B. At the upper levels of the tree the block pointer gives the location of the top element exactly, whereas at the bottom levels the location is only approximate and sequential search inside the block is needed to locate the top element. A navigation pile with buckets of size $\Theta(\log_2 n)$ is important since it gives the same asymptotic time bounds as a normal navigation pile, but it needs only $O(n/\log_2 n)$ extra bits.

Since our goal is different, we propose that, instead of a binary tree, a d-ary tree is used as the underlying tree structure in a navigation pile. We sketch next how this could be done efficiently for $d = 4$; the generalization for larger values of d is also possible.

An offset stored at a branch of a normal navigation pile indicates the top element stored in its leaf sequence. Assume that the offset has γ bits.

The most significant bit of these γ bits tells whether the top element lies in the subtree rooted by the first or second child. In a sense this bit indicates the sorted order of the top elements referred to by the offsets stored at the children. This idea can be generalized by letting the navigation information stored at a branch consist of two parts: a ***state*** and an offset to the top element as earlier. A state should indicate the sorted order of the top elements referred to by the offsets stored at the children in use. For $d = 4$, there are $\sum_{c=1}^{4} c! = 33$ possible states in all.

A complete d-ary tree having 4^η leaves has $(4^\eta - 1)/3$ branches. An offset stored at a branch having height γ should have 2γ bits. Hence, the total number of bits to be stored is

$$\sum_{\delta=0}^{\eta-1} (\lceil \log_2 33 \rceil + 2(\eta - \delta)) 4^\delta < \frac{26}{9} 4^\eta \,.$$

Since the capacity of a 4-ary navigation pile must be a power of 4, the number of bits needed can be no more than 12 times the number of elements stored in the structure.

Let us now consider how the state information can be used. The offsets are used as described earlier so in the *push*() and *pop*() functions the key is to consider how the path updates required by these functions are done. Assume that we are in some branch in a traversal from a leaf to the root. First, we read the state information which also gives the degree of the branch. Second, we unravel whether we came to this branch from the first, second, third, or fourth child. Third, since the old state indicates the sorted order of the top elements referred to by the offsets stored at the children of the branch considered, we can with at most two comparisons determine the position of the top element in its leaf sequence. Finally, the new state and the new offset are computed.

The program carrying out the tasks simply consists of a switch statement having $4 \cdot 33 = 132$ entries. Each entry is specialized for one possible configuration depending on the index of the child from which we came and the old state. Actually, some entries in the switch statement are extraneous since, for example, a branch storing a state that indicates the order of four elements cannot have a degree less than three after a single modification. For $d = 4$ the programs can be written by hand, but for larger values of d they should be generated automatically or semiautomatically. The basic idea is to carry out binary search among the top elements referred to by the offsets stored at the children different from the child, from which we came.

In the case $d = 4$, at most two element comparisons are necessary at each entry of the switch statement. Since the root of a 4-ary navigation pile storing n elements has height $\lceil \log_4 n \rceil$, the total number of element comparisons is the same as that in the binary case. The increase in the arity does not have an effect on the number of element moves performed. Moreover, very few instructions are executed per element comparison, packing and unpacking integers being the main source of instruction overhead.

3.7 Conclusions

Our results are summarized in Table 3.1. Using a heap, which stores pointers or cursors to the elements instead of the elements themselves, similar results could be achieved, but such a data structure would require $\Theta(wn)$ or $\Theta(n \log_2 n + w)$ extra bits. On the other hand, in-place methods seem to require more element comparisons and element moves than those based on navigation piles. Hence, our results fall between these two extremes showing that many element comparisons and element moves can be avoided, even for the dynamic structures, by allowing the usage of $O(n + w\sqrt{n})$ extra bits and $O(\sqrt{n})$ extra elements.

We close the paper with three open problems:

1. A run-relaxed heap [11] can be made to support both the *push*() and *top*() functions in constant time (even though the paper only states a logarithmic bound for the *top*() function). It may be possible to improve the efficiency of our *push*() function in the same way, but it is not clear for us whether such an improvement is of practical value.

2. The iterator validity is discussed in several places in the C++ standard [20]. In the STL a priority queue is not required to support iterators to the elements, but our data structures can be extended to support bidirectional iterators so that the iterators are valid under the insertion and erasure of elements. To achieve this, the element indices are kept in a doubly-linked list, and the addresses of the nodes storing the indices are used as iterators. If each element is associated with its iterator, updates and moves can be handled in constant time. Since there is a direct access from an iterator to the corresponding element, **operator*****() takes constant time. Due to the linked-list structure, **operator++**() and **operator--**() can be performed in constant time.

Table 3.1: Summary of the results.

N : the maximum number of elements stored

n : the current number of elements stored

w : the length of the machine word

#extra bits	$2N+O(w)$			
#extra elements	0			
navigation pile	constructor	*top*()	*push*()	*pop*()
# comparisons	$n-1$	0	$\log_2 \log_2 n+O(1)$	$\lceil \log_2 n \rceil$
# moves	n	0	1	2
# instructions	$O(n)$	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$

#extra bits	$4n+O(w)$
#extra elements	1
sorting	pilesort
# comparisons	$n \log_2 n+0.59n+O(1)$
# moves	$2.5n+O(1)$
# instructions	$O(n \log_2 n)$

#extra bits	$4n+O(w\sqrt{n})$			
#extra elements	$O(\sqrt{n})$			
priority queue	constructor	*top*()	*push*()	*pop*()
# comparisons	$n+o(n)$	0	$2 \log_2 \log_2 n+O(1)$	$\log_2 n+\log_2 \log_2 n+O(1)$
# moves	n	0	1	2
# instructions	$O(n)$	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$

#extra bits	$4n+O(w\sqrt{n})$			
#extra elements	$O(\sqrt{n})$			
priority deque	constructor	*top*() *bottom*()	*push*()	*pop_top*() *pop_bottom*()
# comparisons	$1.5n+o(n)$	1	$4 \log_2 \log_2 n+O(1)$	$2 \log_2 n+2 \log_2 \log_2 n+O(1)$
# moves	n	0	2	5
# instructions	$O(n)$	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$

Since for our data structures each modifying operation requires only a constant number of element moves, the overhead of maintaining the element indices for the iterators is a constant per operation. However, we do not know how efficiently random-access iterators can be supported.

3. If the navigation pile is extended to allow access to the elements through iterators, operations *increase_key*() and *erase*() can be supported with the same efficiency as *push*() and *pop*(), respectively. Since a run-relaxed heap supports the *increase_key*() operation in constant time, one could ask whether our resource bounds for that function could also be improved.

Chapter 4

Local heaps

4.1 Introduction

A ***priority queue***, with respect to an ordering function *less*(), is an abstract data structure which stores a collection of elements and provides the following operations:

push(x): Insert element x into the data structure.

top(): Return the reference to the ***top element***, i.e. to an element x such that for any other element y present in the priority queue *less*(x, y) returns false.

pop(): Erase the top element from the priority queue.

A d***-ary heap***, which is an efficient way to implement a priority queue, is a left-complete d-ary tree, in which each node stores one element and which is represented in a sequence by storing elements in breadth-first order A d-ary heap, with respect to an ordering function *less*(), has the following crucial property: for each branch of the tree, the element y stored at that node is no smaller than the element x stored at any child of that node, i.e. *less*(y, x) must return false. The binary heaps were invented by Williams [38] and the generalization to $d > 2$ was suggested by Johnson [23].

Traditionally, algorithm designers analyse the asymptotic number of primitive operations executed. For example, when referring to a heap data structure, the operations considered in a standard analysis are element comparisons and element moves. For many years this approach has dominated the

development of algorithms and data structures. This approach has been shown to be insufficient and has led to evident deviations between the predicted performance and the experimental results, mainly because of reasons related to architectural details like pipelining, register spilling, and caching (see, for example, [3, 26]). Of these, caching effects are the most important.

Usually computer memory is composed of different random-access storage levels, abiding the principle of inclusion: the presence of a datum at level i implies the presence of the same datum at level $i+1$. Level i is assumed to be faster and smaller than level $i+1$. Each level is divided into disjoint and consecutive ***blocks*** of the same size, but block dimensions at different levels can vary. With the term ***cache*** we indicate the levels bridging main memory and the CPU. The cache stores the copy of a fraction of the data held in main memory that is likely to be requested by the CPU. When a data request cannot be satisfied by the cache, i.e. in the case of a ***miss***, a contiguous block of memory containing the data is copied from main memory to the cache, according to the replacement strategy adopted. We assume that when a miss occurs the block is copied from main memory to the cache following the ***least-recently-used*** (***LRU***) block-replacement policy. Furthermore we assume that the cache is ***fully-associative*** so that the block can be placed to any cache location. For further details on the architecture of computer memories, we refer to [19].

Since in most contemporary computers an access to main memory (and a block transfer from it) is more expensive than other primitive operations, caching effects cannot be ignored in the design of algorithms and data structures. A ***cache-aware*** algorithm needs to know the memory characteristics of the computer on which it is executed.A ***cache-oblivious*** algorithm, as defined in [15], is able to reach a good cache behavior without knowing architectural details of the cache.

Our data structures are cache aware, and needs to set a parameter h in order to improve the cache performances.

In this paper we present and evaluate experimentally a memory layout for binary heaps, which has a good spatial locality, thereby using the memory structure efficiently and in addition to this having a small computational overhead. The basic idea of our data structure is to locally group the elements stored in a binary heap. The number of elements present in such groups is selected to be $2^{h+1}-1$, where h is an integer determining the degree of localization of the data structure. This kind of localization technique is motivated by the memory structures of the modern computers. In spite of its

simplicity, our data structure is able to obtain a good worst case performance with respect to the number of element comparisons and element moves performed. Moreover, the data structure still operates in-place without requiring any extra memory.

Let n be the number of elements stored in the data structures. If n is the total number of elements, in particular, the *push*() worst case requires $\log_2 \log_2 n+O(1)$ element comparisons, $\log_2 n+h+O(1)$ element moves, and $(2/(h+1)) \log_2 n+O(1)$ cache misses, *top*() is executed in constant time, and the *pop*() worst case requires $(1+1/(h+1)) \log_2 n+O(h)$ element comparisons, $\log_2 n+O(h)$ element moves, and $((2/(h+1)) \log_2 n+O(1)$ cache misses. In the design of this heap layout, the hierarchical memory structure, which is a basic element of the majority of computers, assumes a fundamental importance.

This definitions seem to state that cache oblivious algorithms are in general to be preferred to cache aware algorithms. However, in the attempt of get a good cache use, several cache oblivious algorithms and data structures involve the execution of so complicated sequences of additional instructions, e.g. sophisticated index manipulations, that the number of operations considered significant by the traditional increases in a considerable way, worsening the general performances. For example, in the applications of the layout for the cache oblivious search trees discussed in [31] is analysed the well known heap data structure in which the locations of the nodes are joined together with a global recursive technique (if n is the number of elements $O(\log \log n)$ levels of recursions). In spite of the good use of the cache and the complete independence from the architecture, the employment of loops with a non-constant number of iterations inside the function code for the calculation of node indices makes the data structure only of theoretical interest. Moreover, *pop*() in the worst case requires many element comparisons and element moves $(2 \log_2 n+O(1))$, whereas $\log_2 n+\log_2^* n+O(1)$ comparisons are sufficient for deleting a minimum element from a binary heap (Gonnet and Munro [17], corrected by Carlsson [6]).

The remainder of the report consists of five sections. In Section 4.2 we describe the data structure and we analyse the performances of the main operations, comparing them with some classical heap versions. In Section 4.3 we explain the implementation techniques that we employed in the programming phase. In Section 4.4 we describe an alternative realization of the priority queue able to reduce the overhead for navigate the data structure. In Section 4.5 we show the experimental results relative to our implementations and the conclusions are presented in Section 4.6 .

4.2 Design and analysis of the data structure

We will now briefly recall the principal variants of the algorithms for the execution of the main operations on the classical heap data structures. We will also show the results known for the asymptotic analysis of the relative worst case performances for what concerns the number of element comparisons, element moves, and cache misses.

All the data structures analysed in the paper require $O(\log n)$ arithmetic operations executing *pop*() and *push*() and $O(1)$ arithmetic operations for executing *top*(). $A[i]$ will denote the $(i+1)$-th element stored in the heap. We will use the following parameters when evaluating the performances of the operations analysed:

n: number of elements currently present in the heap.

M: cache capacity.

B: block size, which it is assumed to be a power of 2.

T: element size, which, for the sake of simplicity, we will assume to be a power of 2 not greater than B.

k: number of elements that can be contained in one block, i.e. B/T.

d: arity of the heaps, that we assume to be a power of 2.

M, B, and T are measured in bytes. We will assume that no element span cache blocks. Since when accessing to an element it is often necessary to access to all its sibling locations, it would be useful that all the children of any given element to be present in the same block. In this case we would say that the heap is well-aligned. In following analysis we will assume that the heaps are always aligned in the worst possible way, i.e. that the alignment is such that maximize the number of cache misses occurred.

We also assume that the reader it is familiar with binary heap implementation and analysis. For specific details we refer to [9].

4.2.1 *push*(x)

Bottom-up execution

The path connecting the location $A[n]$ and the root is traversed in a bottom-up manner, copying every element to its child location until the first element

equal or greater than x is found or until the root is reached. The new element is copied into the child location of this element.

Binary heap

In the worst case it is necessary traverse up the heap. The number of both element comparisons and element moves required is therefore $\log_2 n+O(1)$. We need to access $\log_2 n+O(1)$ elements belonging to different blocks. This implies that the number of cache misses is trivially $\log_2 n+O(1)$.

Multiary heap

Analogously to the binary case the heap is traversed, requiring $\log_d+O(1)$ element comparisons, element moves, and cache misses.

Binary search execution

We look for the correct index j of the new element with a binary search on the path connecting the location $A[n]$ with the root and we copy each element on the path connecting the parent of $A[n]$ with $A[j]$ to its child position, in order to make place to the element to be inserted. We will assume that the cache is big enough for contain all the elements on the path from any leaf to the root.

Binary heap

This operation requires $\log_2 \log n+O(1)$ comparisons and $\log_2 n+O(1)$ element moves, since the element inserted can be greater than the root. All the elements on the path from the new element until the root can need to be visited, so that the number of cache misses involved is $\log_2 n+O(1)$.

Multiary heap

The number of comparisons required is $\log_2 \log n+O(1)$ and the number of moves and cache misses is $\log_d n+O(1)$.

4.2.2 *top*()

The reference to the root element is returned. For both the binary and d-ary case no element comparison, element move is required.

4.2.3 *pop*()

Top-down execution

We traverse top-down the heap following the greatest child of each element visited. During this traversing we copy each element on that path to its parent location, until the first element $A[j]$ not greater than $A[n-1]$ is found; then we move the last element to the parent location of $A[j]$.

Binary heap

In the worst case it is necessary traverse down the heap. The traversing requires $2\log_2 n+O(1)$ element comparisons and $\log_2 n+O(1)$ element moves. The number of cache misses involved is $2\log_2 n+O(1)$. In order to prove this fact it is enough to consider the following cases. If $k = 1$ or $k = 2$ and the two elements $A[1]$ and $A[2]$ belong to different blocks the result is trivially true. We consider now the case in which k assumes a greater value. If $A[k-3]$ and $A[k-2]$ belong to different blocks all the elements on the rightmost path from $A[k-2]$ until the bottom of the heap will have a sibling occupying a different block. If the path traversed during the *pop*() execution is the rightmost one we will get the result.

Multiary heap

In the worst case it is necessary to traverse down the heap. The execution requires $d\log_d n+O(1)$ element comparisons and $\log_d n+O(1)$ element moves. If $k = 1$ the number of cache misses required is trivially $d\log_d n+O(1)$, under the assumption that M is at least equal to $(d+2)B$. If k is greater than 1 but not greater than d and the leftmost k elements of any group of siblings belong to two blocks, the number of cache misses is $(d/k+1)\log_d n+O(1)$ (assuming that M is at least equal to $(d/k + 3)B$). If k assumes a greater value, let α be the rightmost group of siblings of the last complete level having less than k elements. If α spans two cache blocks all the group of siblings on the rightmost path of the heap from α would span the cache blocks too. This implies that the number of cache misses is $2\log_d n+O(1)$, under the assumption that M is at least $4B$.

Bottom-up execution

The index l of the bottom element $A[l]$ whose each ancestor is the greatest element among their siblings is searched traversing top-down the whole

heap. The heap is traversed up starting from $A[l]$ searching the bottommost element $A[j]$ not smaller than $A[n-1]$. Each element on the path from the root (excluding it) to $A[j]$ is copied to its parent location and $A[n-1]$ is copied to the $A[j]$ location. We will assume that M is such to make the cache able to contain all the elements blocks of any path from the bottom to the root and their sibling blocks; if k is equal to 1 its value is at least $d\lceil\log_d((d-1)n+1)\rceil-d+2$, if k is greater than 1 but not greater than d it is at least $(d/k+1)\lceil\log_d((d-1)n+1)\rceil-d/k+1$ and if k is greater than d it is at least $2\lceil\log_d((d-1)n+1)\rceil$.

Binary heap

In the worst case $A[n-1]$ is moved to the first level of the heap, requiring $2\log_2 n+O(1)$ element comparisons and $\log_2 n+O(1)$ element moves. The cache analysis is the same as for the top-down version.

Multiary heap

The worst case is similar to the binary variant, requiring $d\log_d n+O(1)$ element comparisons and $\log_d n+O(1)$ element moves. The cache analysis is the same as for the top-down version.

Multiary heaps are able to get a better cache behavior compared with the binary heaps [26]. However the number of comparisons increases quickly with the value of d. This fact motivated us in the search of an improvement of the binary heap layout in order to limit the number of comparisons reducing at the same time the number of cache misses for the worst case. We called this kind of heap h-local heap.

4.2.4 Local heap description

A ***h-local heap*** is a (2^{h+1})-ary heap in which each node stores a binary heap of size $2^{h+1}-1$. We will call the nodes of a h-local heaps *fat-nodes* and the number of elements present in a full fat-node *fatness*, that we will denote by the parameter F (depending therefore only on h). If the value of h is not relevant we will refer to this data structures calling it simply *local heaps*. The last fat-node is the only one that may store a binary heap of size less than $2^{h+1}-1$.

Let call *in-node heap* the heap contained in any fat-node. Let α be a fat-node and x the i-th (from the left) element of the bottom level of the

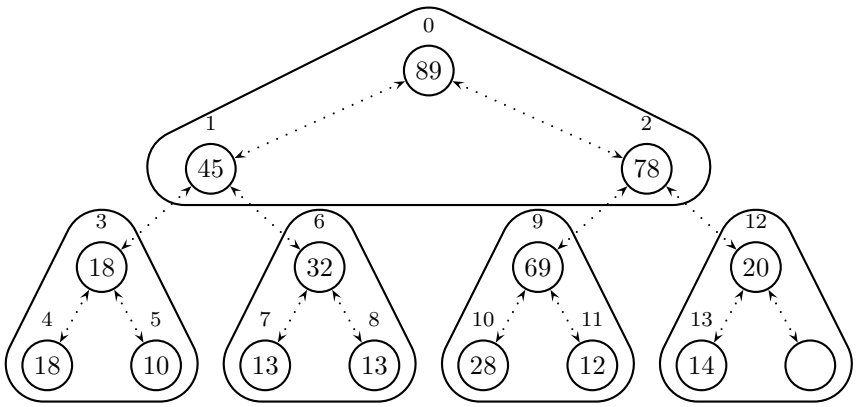


Figure 4.1: A 1-local heap of size 14.

in-node heap contained in α. Let c_l and c_r be the elements stored at the roots of the the in-node heaps contained respectively in the $(2i-1)$-th and $(2i)$-th child of α. The final constraint simply requires that x is never smaller than both c_l and c_r. It appears clear that a 0-local heap is a normal binary heap.

From this definition is not difficult to see that the main idea in the design of this type of heaps is simply to use a classic heap changing the index of the nodes in order to localize groups of nodes having similar depth value.

Having the index of an element, we can calculate the index of its parent and its children in the following way:

node_index *parent*(node_index i) **const**;

Effect: fat-node_index $j \leftarrow \lfloor i/F \rfloor$;
if $(i \neq jF)$
return $\lfloor (i+(jF-1))/2 \rfloor$;
else
return $\lfloor (j+(F-1)\lfloor (j-1)/(F+1) \rfloor + F-2)/2 \rfloor$;

node_index *left_child*(node_index i) **const**;

Effect: fat-node_index $j \leftarrow \lfloor i/F \rfloor$;
if $(i < \lfloor F/2 \rfloor + jF)$
return $2i - jF + 1$;
else
return $F(2i + (1-F)j - F + 2)$;

node_index *right_child*(node_index i) **const**;

Effect: fat-node_index $j \leftarrow \lfloor i/F \rfloor$;
if $(i < \lfloor F/2 \rfloor + jF)$
return $2i - jF + 2$;
else
return $F(2i + (1-F)j - F + 3)$;

In order to prove the correctness of these formulas we show the reasoning steps for the calculation of *left_child*, leaving to the reader the verification of the other two (built in an analogue way). Fixed a index i of an element x belonging to a fat-node α we can calculate the index left child y of x in two different way according to the fact if x is in the bottom level of α or not.

If x is in the bottom level of α we need to find the fat-node index of α (we think to fat-nodes as elements of a 2^{h+1}-ary heap numbered in breadth-first order), which is given by $\lfloor i/F \rfloor$. If y belongs to the k-th fat-node child of α, the left child index of x will be given by $(2^{h+1} \lfloor i/F \rfloor + k)F$. If i_α is the index of x inside of α, given by $i - \lfloor i/F \rfloor F$, k will be $2(i_\alpha - \lfloor F/2 \rfloor) + 1$. Putting all together with simple calculations we get the formula in the second if-branch of *left_child*.

If x is not in the bottom level of α, the index j_α inside α of y will be given by $2i_\alpha + 1$. *left_child*(i) is then equal to $\lfloor i/F \rfloor F + j_\alpha$, equal to the formula present in the if-first branch of *left_child*.

Using these formulas it is easy to implement the main operations on these data structures. *push()* can be executed like for an ordinary binary heap, as explained before. *top()* simply returns a reference to the first element of the root fat-node. *pop()* can be executed traversing top-down the fat-node heap.

We start from the top element location of the h-local heap and, traversing top-down the data structure, every time we visit the root of a in-node heap contained in a fat-node we execute the following operations:

1) we traverse top-down the in-node heap copying, to the location of each element on the path traversed, its greatest child (until we arrive to the bottom-level of the fat-node).

2) we compare the greatest child c of the last element copied with $A[n-1]$ (currently positioned as the last leaf in the last fat-node of the data structure);

- if $A[n-1]$ is not the biggest we simply copy c to its parent position and we continue the traversing

- otherwise, since we know now that the correct position of x is inside the last fat-node modified, we traverse it bottom-up copying every element smaller than $A[n-1]$ to its child position on the path traversed. We terminate inserting the last element $A[n-1]$.

4.2.5 Local heap analysis

We now analyze the worst case performances of the h-local heap in term of number of element comparisons, element moves, and cache misses. For this last element, we need some preliminary considerations. First of all we calculate the maximum number of cache misses necessary to traverse a fat-node, varying the parameters k and h.

$k \leq 4$: the worst case number of cache misses is $h+1$. This follows from the fact that a worst case alignment occurs when the root of the fat-node and its left child belong to different blocks.

$k > 4$ and $h \geq \log_2 k$: the worst case number of cache misses is reduced to $h-\log_2 k+3$ because consecutive elements on the rightmost path of the fat-node belong to the same blocks.

$h < \log_2 k$: the worst case number of cache misses is trivially 2.

It is also important to consider that if α is the fat-node on the rightmost path of the h-local heap belonging to the last fat-node full level having less than k elements, all the fat-node on the rightmost path from α will span the blocks in the same way.

push()

The bottom-up execution requires not more than $\log_2 n+h+O(1)$ element comparisons and element moves. If $k \leq 4$ the number of cache misses is not more than $\log_2 n+h+O(1)$; in the case in which $k > 4$ and $h \geq \log_2 k$ the number of cache misses is not more than $((h-\log_2 k+3)/(h+1))(\log_2 n+h)+O(1)$; if finally $h < \log_2 k$ the number of cache misses becomes $(2/(h+1)) \log_2 n+O(1)$.

If the binary search is performed along the path from the location following the last leaf until the root $\log_2 \log n+O(1)$ element comparisons and not more than $\log_2 n+h+O(1)$ element moves are required. We assume that M is at least $(\lceil \log_2(n+1) \rceil +h)B$. The number of cache misses is the same as for the bottom-up execution.

top()

The execution of this operation does not require any element comparisons nor element moves.

pop()

The traversing of a fat-node requires $h+2$ comparisons. Since the number of fat-node on the path from any leaf until the root is $(\log_2 n)/(h+1)+O(1)$, $(1+1/(h+1)) \log_2 n+O(h)$ element comparisons and $\log_2 n+O(h)$ element moves are executed. The worst case number of cache misses varies according to the parameters h and k. The worst case number of cache misses involved considering a single in-node heap is obtained if the last two elements belong to different blocks, and it is necessary to traverse it on its rightmost path. The root of such in-node heap contained in a fat-node on the rightmost path and the root of the in-node heap contained in its sibling fat-node belong to different blocks if and only if $h \geq \log_2 k-1$. It is now easy to see that the number of cache misses is:

$k = 1$: $2 \log_2 n+O(h)$.

$k = 2$: $(2-1/(h+1)) \log_2 n+O(h)$.

$h < \log_2 k-1$: $(2/(h+1)) \log_2 n+O(1)$.

$h \geq \log_2 k$ and $k > 2$: $((2h-2 \log_2 k +4)/(h+1)) \log_2 n+O(h)$.

$h = \log_2 k-1$: $(3/(h+1)) \log_2 n+O(1)$.

In the alternative execution employing the binary search along the path following the greatest child of each element for find the correct place of $A[n-1]$, the number of element comparisons becomes $\log_2 n + \log_2 \log n + O(h)$ and number of element moves is $\log_2 n + h + O(1)$. We assume that M is at least $(2\log_2 n + 2h + O(1))B$; the number of cache misses is again the same executed by the bottom-up *push*().

4.3 Implementation details

In order to get good performances, in the implementation phase we took into account several elements. Basically, employing the language C++, we produced, for *pop*() and sorting execution, an implementation for each value of h that we considered interesting (from 1 to 5), reducing the number of arithmetic operations executed and the memory accesses. We considered also the registers use (avoiding to employ a big number of variables). We avoided the use of a template parameter h in a unique implementation because a specialized implementation for every value of h let us to augment the code optimization level. The code analyzed in this section is the result of numerous optimization steps in which we evaluated the relative performances testing our programs. We also implemented a bottom-up version for *pop*() and sorting. In this section we analyze the code relative to the (top-down) *pop*() execution for $h = 3$ (shown in the appendices), because it is enough for explain in detail all the implementation techniques that we employed.

All the code is divided in three parts:

heap policy: it contains the set of basic functions useful in every contests, like *parent*(), *last_leaf*() or *first_child*().

utility functions: this part of the code implements the functions useful for the implementation of the following group of functions. *sift_down*() and *sift_up*(), that percolate an element in the heap (down and up respectively), are part of this group.

heap functions: this part of the implementation includes all the functions that can be called for execute the main operations on the heap. *pop_heap*(), *make_heap*(), and *sort_heap*() belong to this group.

In Appendix A.1 we show some of the policy function necessary in the *pop* implementation. We leave to the reader the comprehension of details

(like template parameters) that are obvious or not important. The function *first_child*() simply returns the index of the first child of the element whose index is passed as parameter. It is easy to see that it uses the formulas relative to the $left\_child()$ function showed in the Section 4.2. In order to speed up the execution it stores the result of some arithmetical calculations in some variables that can easily be reused inside the rest of code. *top_all_present*() and *top_some_absent*() returns the index of the maximum child of the element whose index is passed as parameter (if all the children are present or if some children are missing).

We pass now to the implementation code of *pop*(). In Appendix A.2 we show the code for the utility function *sift_down*(), that percolates down in the heap the element passed as parameter.

The code of this function is constituted of four parts. The first one is the core of the implementation, consisting of a while-loop with all the instructions necessary to traverse a fat-node. Since at the end of the traversing of the in-node heap, it may be necessary to access the location of everyone of the children of the bottom elements, first of all we calculate the index of the root of the last fat-node α having all the children. Even if it would have been possible to use a more simple while-loop employing the policy functions necessary for the traversing (like *first_child*()), we decided to unroll the loop for the traversing of every single fat-node for optimization questions. For example, traversing the fat-node in the unrolled loop we do not need to know, in order to find the *first_child*() of a given element x, if it belongs to the bottom level of its in-node heap, since we always know which in-node heap level we are processing.

We used a variable m containing the index of the root of the fat-node following α for the while condition. Besides m we need to use other three index variables. k stores the value of the root of the fat-node, that will be useful at the end of the in-node traversing in order to avoid some arithmetical operations. i and j are important for navigate the fat-node node. We use the variable ak storing the value of the address of the the in-node heap root. using ak we avoided some additions necessary for access its elements during the execution of element comparisons and element moves, replacing $a[k+j]$ $(*(a+k+j))$ with $*(ak+j)$, where j represent the index of the element inside its in-node heap.

Lines 021 and 022 calculates the index of the left children (root of a child fat-node) of the element whose index value is i. On 024 we have the if-else

tree necessary for find correct location of element x (in the case in which it is inside the fat-node traversed). With this if-else tree we traverse up the in-node heap and moving back the elements modified previously.

In the most part of the cases the value of i at the end of the while-loop is the index root of a full in-node heap contained in a fat-node having no children. Since this situation is easy to manage, we first check if this is really the case on line 045, in which the second part of the code starts. This part is very similar to the first one, except for the fact that the element to sift down x is involved for each level of the in-node heap traversed. In the case in which i is the index of an element root of a fat-node having at least one child or a non-full in-node heap, the second part of the code is not executed, whereas the third part begins. In the third part the rest of the heap is traversed involving x with one comparison for every level. In order to reduce the number of index operation for the rest of the code, the third part checks for every loop execution if the element whose index is i has both the children. After the third part we only need to continue the traversing being careful to the possibility that the element whose index is i has not all the children.

The code of the third and the fourth part employ the policy functions above described and it is not very optimized. It executes only a constant number of instructions (depending only on h) and in the most part of the *pop*() executions it is never processed.

In the Appendix A.3 we show a function that is part of the heap functions and executes *pop*() employing *sift_down*().

The complete code of the local heap programs is accessible via the home page of the CPHSTL project [10].

4.4 An alternative realization

Analyzing the formulas used in the *parent*(), $left\_child()$, and $right\_child()$ functions showed in the Section 4.2, it easy to see that a remarkable part of the cost in term of time will concern the execution of divisions and multiplications of index values by F, the number of elements in a full fat-node.

The main problems derives from the fact that the value of F is never a power of two (otherwise, assuming that *left shift*, *right shift*, bitwise *and*, *or*, and *not* instructions executable in constant time, we could implement all the operations of the data structures more easily) and the degree of a fat-node is not equal to F (but it is $F+1$, a power of two).

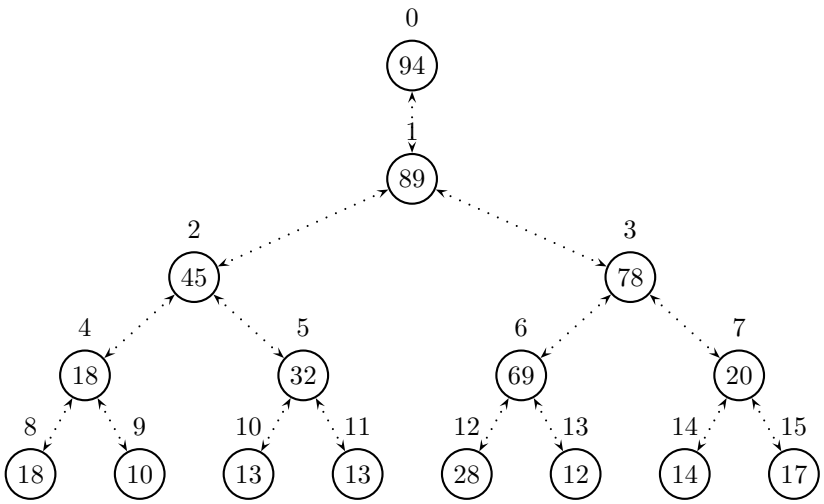


Figure 4.2: A complete extra-rooted heap.

In order to solve this problems and accelerate the execution of all the operations, it could be possible to consider an alternative kind of the fat-nodes. In this new approach, every fat-node will contain a modified heap that we will call *extra-rooted heap*. This kind of heap is simply composed by an ordinary binary heap in which the root will be (in breadth-first order) instead of the first, the second element of the whole structure, whereas the first location will be occupied by an element not smaller than any other element. We omit the formal definition about the relations among elements contained in different fat-nodes of this type, analogue to that one given in the Section 4.2.

The employing of extra-rooted heaps for the h-local heaps makes equal the value of the fatness to the degree of a fat-node (now therefore both power of two). This fact let us to simplify the implementation of the basic functions used in a massive way in the implementation of all the operations.

The height of the total data structures, seen as trees of elements, grows making the number of element moves equal to $(1+1/(h+1))\log_2 n+O(1)$. In spite of the consequent increasing of the number of element moves for some operation, this new kind of fat-node realization could represent, if the element comparisons and the element moves are not too expensive, a better approach from a practical point of view.

4.5 Experimental results

We tested our implementations for these data structures against the bottom-up Silicon Graphics heap implementation using different elements types for both the normal top-down and bottom-up approach and the top-down extra-rooted fat-node realization, for h={1, 2, 3, 4, 5}.
The element types that we considered are:

integers : unsigned integers; with this element type comparisons and moves are cheap.

integers with logarithmic comparison : unsigned integers using an ordering function comparing the logarithm of the two integer arguments.

big integers : unsigned integers represented as strings of digits.

big integers with logarithmic comparison : big integers using an ordering function comparing the logarithm of the two integer arguments.

This way we analysed results for both cheap and expensive element moves and/or comparisons.
In this paper we decided to show and analyse the performances for sorting because considered a kind of test letting us to note the most significant difference about the speed of the data structures.

As showed in the pictures, when we deal with a not too small number of elements, the performances for all the approach, except for integers with logarithmic comparison (and for the extra-rooted heap with logarithmic comparisons), result better than those relative to the classic binary heap, especially for $h = 3$ and $h = 2$.
The approach employing extra-rooted fat-nodes results slightly better for integers than the top-down normal approach, is spite of the growing of the height of the heap. For the dimension of the fat-nodes, the value $2^{h+1}-1 = 7$ or $2^{h+1}-1 = 15$ ($2^{h+1} = 8$ or $2^{h+1} = 16$ for the extra-rooted version) is therefore

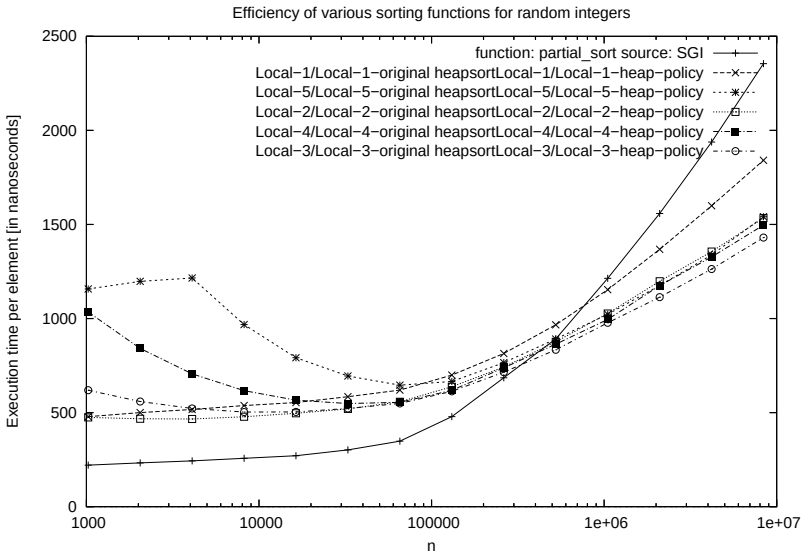


Figure 4.3: Sorting test for the h-local heaps and Silicon Graphics implementation (integers).

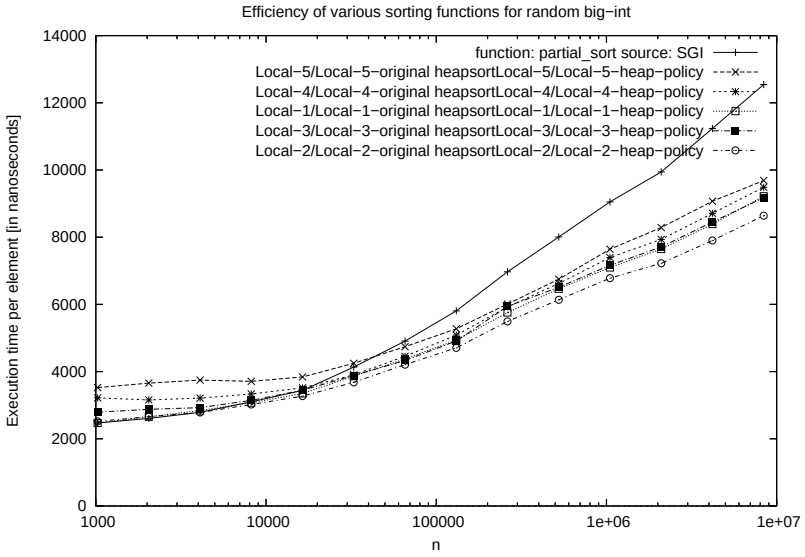


Figure 4.4: Sorting test for the h-local heaps and Silicon Graphics implementation (big integers).

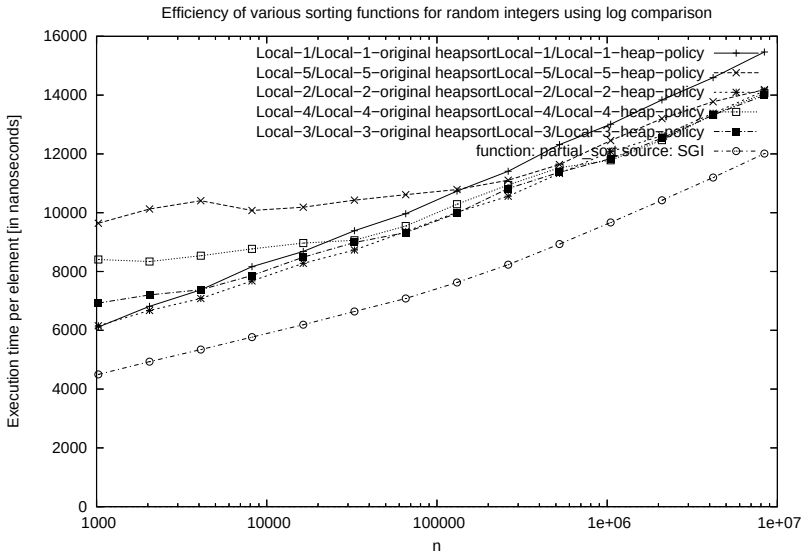


Figure 4.5: Sorting test for the h-local heaps and Silicon Graphics implementation (integers with logarithmic comparisons).

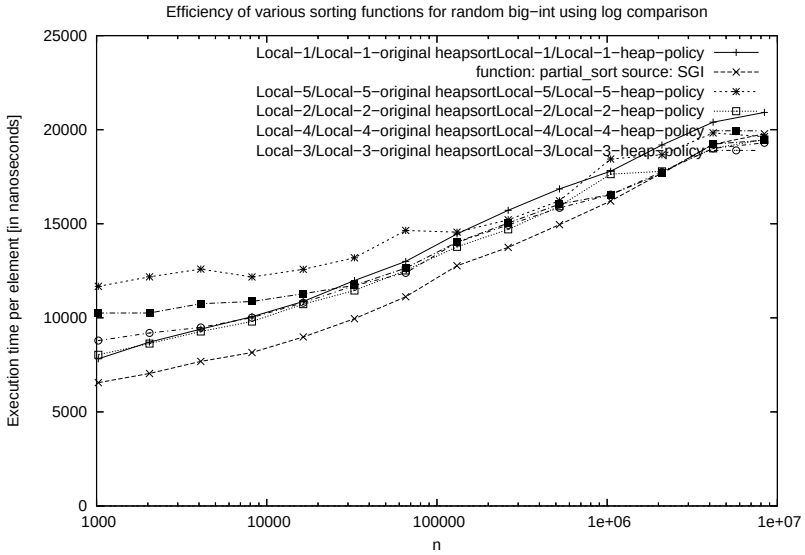


Figure 4.6: Sorting test for the h-local heaps and Silicon Graphics implementation (big integers with logarithmic comparisons).

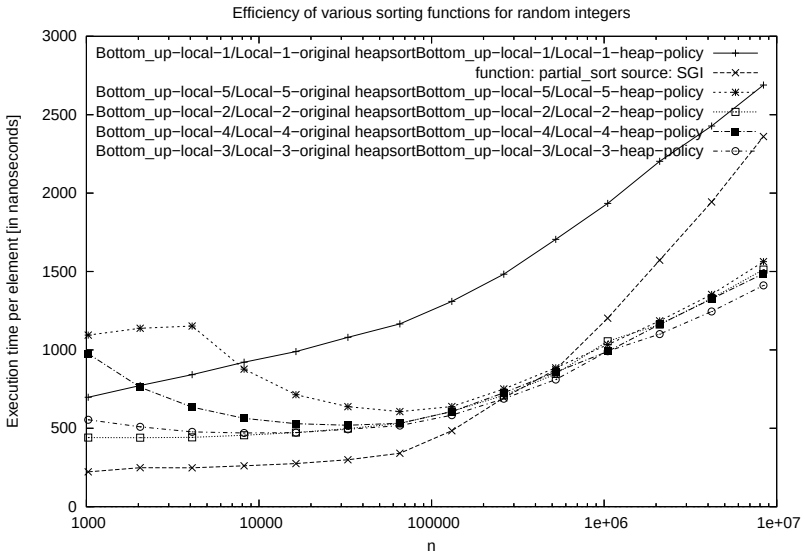


Figure 4.7: Sorting test for the h-local heaps and Silicon Graphics implementation with bottom-up approach (integers).

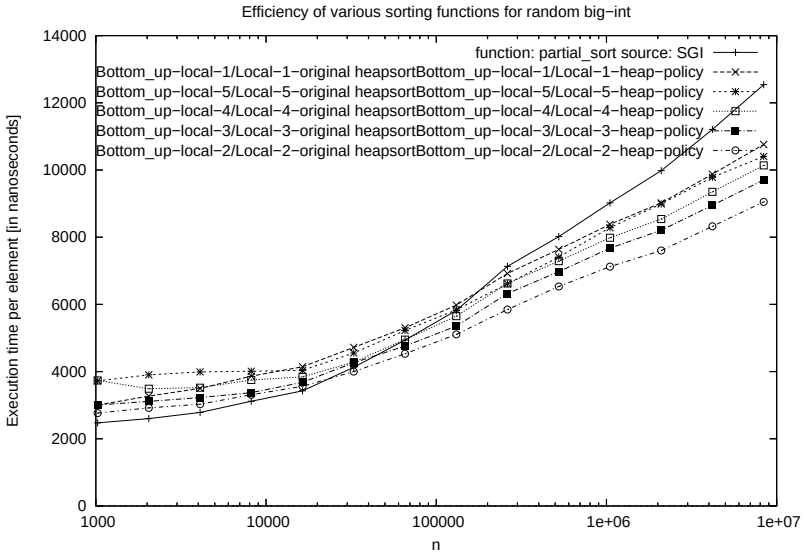


Figure 4.8: Sorting test for the h-local heaps and Silicon Graphics implementation with bottom-up approach (big integers).

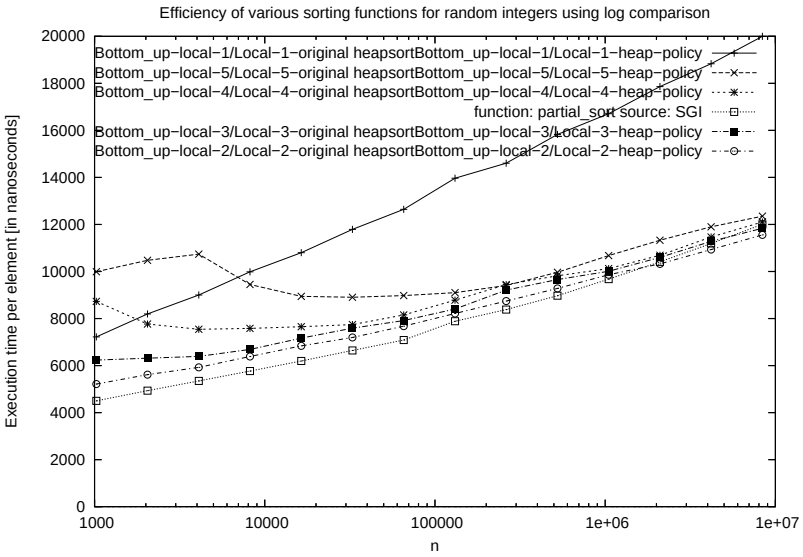


Figure 4.9: Sorting test for the h-local heaps and Silicon Graphics implementation with bottom-up approach (integers with logarithmic comparisons).

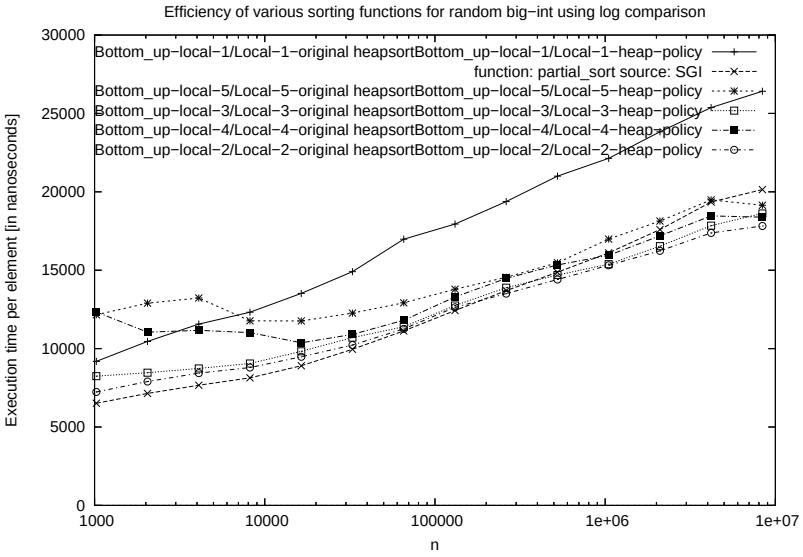


Figure 4.10: Sorting test for the h-local heaps and Silicon Graphics implementation with bottom-up approach (big integers with logarithmic comparisons).

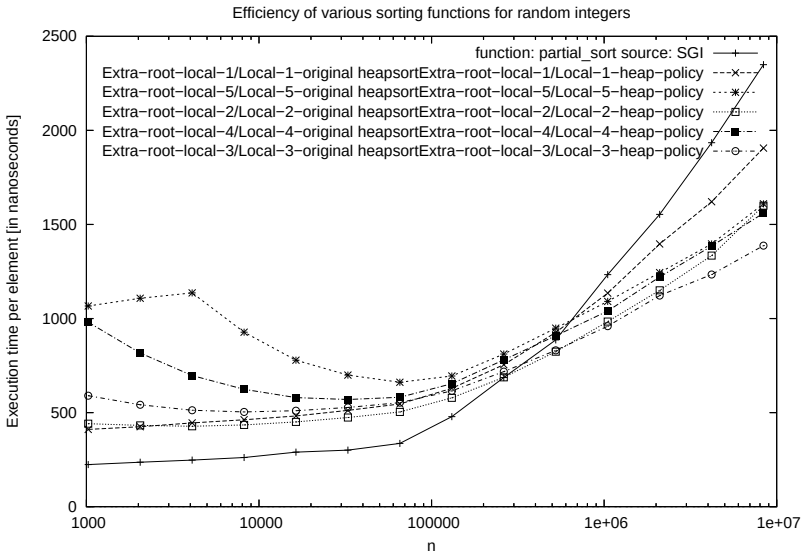


Figure 4.11: Sorting test for the extra-rooted h-local heaps and Silicon Graphics implementation (integers).

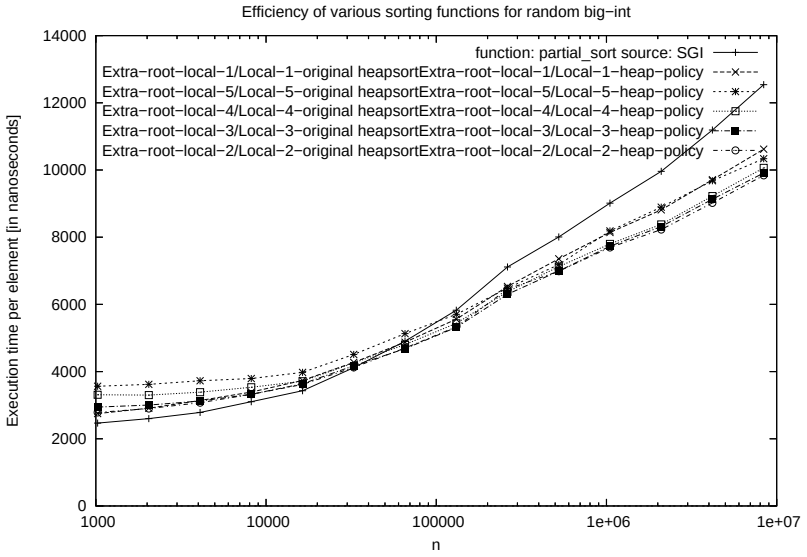


Figure 4.12: Sorting test for the extra-rooted h-local heaps and Silicon Graphics implementation (big integers).

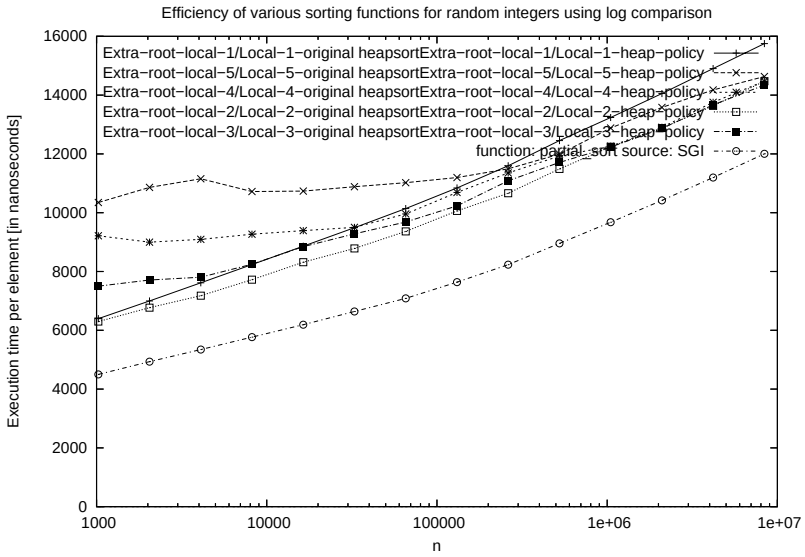


Figure 4.13: Sorting test for the extra-rooted h-local heaps and Silicon Graphics implementation (integers with logarithmic comparisons).

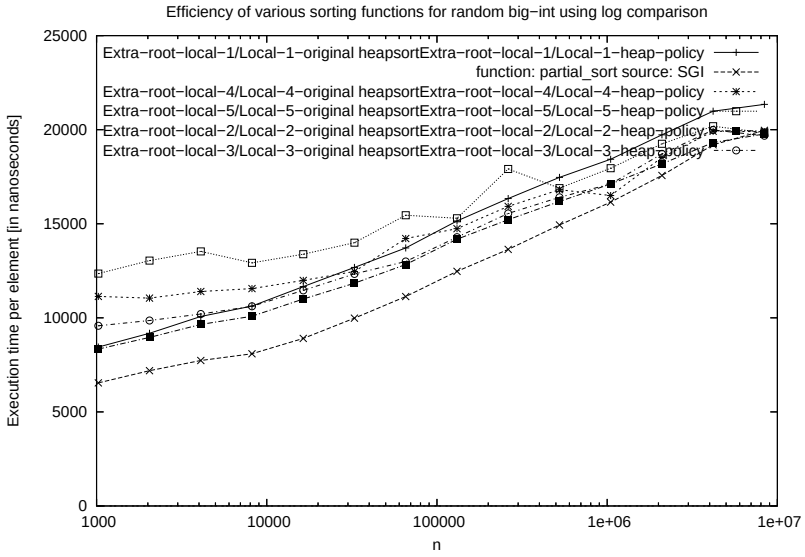


Figure 4.14: Sorting test for the extra-rooted h-local heaps and Silicon Graphics implementation (big integers with logarithmic comparisons).

enough to get the best performances on the computer on which we tested our implementations (AMD Athlon(tm) running with 1032.536 CPU MHZ and 256 KB of cache size).

4.6 Conclusions

We showed how it can be possible to improve the performances of in-place heaps with a very simple localization technique taking into account the cache use and, at the same time, with good performances even from a theoretical point of view.

For the moment, we tested the use of this technique only on binary heaps, but it comes natural to think that we could improve the performances of multi-way heaps in a analogous way; for example, 3-way or 4-way heaps could be interesting for this purpose because we could improve the cache use (for an appropriate value of h) being sure that the number of element moves for the *pop*() execution would be reduced and the number of element comparisons would not increase too much.

We concluded with a natural question: could it be possible to improve our heap data structures employing a more sophisticated localization technique able to work well for every kind of real architecture?

Chapter 5

Conclusions

In this thesis we showed how it is possible to build a priority queue, the navigation pile, making the worst case number of element comparisons, element moves and instructions close to the absolute minimum using at the same time only a linear number of extra bits.

We also demonstrate how a simple localization technique can be useful for improve significantly (when the number of elements is not too small) the performances of binary heaps without using extra memory.

The basic concepts underlying both the papers are simple, easy to understand and at the same time powerful. For this reason we think that the main ideas of the two articles could be easily presented in any modern textbook of data structures.

A number of element comparisons and element moves close to the absolute minimum (for *pop*() and sorting applications) joined with a non superlinear number of extra bits seems to be inextricably connected to a considerable number of cache misses for every priority queue. Therefore, we conclude with a natural question related to both the articles presented in the thesis:

How the localization technique used in the article "Experimental evaluation of local heaps" could improve the performances of the implementation of a navigation pile employing it for the offset tree?

Appendix A

Local heap code

A.1 Policy function code

```
001 template <> template <typename position, typename ordering>
002 typename heap_policy<position, ordering>::index
003 heap_policy<position, ordering>::last_leaf () const {
004    return n - 1;
005 }

006 template <> template <typename position, typename ordering>
007 typename heap_policy<position, ordering>::index
008 heap_policy<position, ordering>::first_child (
009    heap_policy<position, ordering>::index i
010 ) const {
011    index k = (i / 15);
012    index m = k * 15;
013    if((i - m) < 7) return (i << 1) - m + 1;
014    else return (30 * i) - 195 - (210 * k);
015 }

016 template <> template <typename position, typename ordering>
017 typename heap_policy<position, ordering>::index
018 heap_policy<position, ordering>::top_all_present (
019    position a,
020    heap_policy<position, ordering>::index i,
```

```
021    const ordering& less
022 ) const {
023    typedef typename heap_policy<position, ordering>::index index;
024    index k = (i / 15);
025    index m = k * 15;
026    if((i - m) < 7) {
027       m = (i << 1) - m + 1;
028       k = m + 1;
029    }
030    else {
031       m = (30 * i) - 195 - (210 * k);
032       k = m + 15;
033    }
034    return less(a[m], a[k]) ?  k :  m;
035 }

036 template <> template <typename position, typename ordering>
037 typename heap_policy<position, ordering>::index
038 heap_policy<position, ordering>::top_some_absent (
039    position a,
040    heap_policy<position, ordering>::index i,
041    const ordering& less
042 ) const {
043    return first_child(i);
044 }
```

A.2 sift_down() function code

```
001 template <typename position, typename index, typename element,
002    typename ordering, typename policy>
003 void
004 sift_down(position a, index i, element x, const ordering& less, policy&
p) {
005    index j;
006    index m = 15 * (p.last_leaf() / 240);
```

```
007   while (i < m) {
008      index k = i;
009      position ak = a + i;
010      i = (less(*(ak + 1), *(ak + 2))) ?  2 :  1;
011      *(ak) = *(ak + i);
012      j = i;
013      i = (i << 1) + 1;
014      if(less(*(ak + i), *(ak + i + 1))) ++i;
015      *(ak + j) = *(ak + i);
016      j = i;
017      i = (i << 1) + 1;
018      if(less(*(ak + i), *(ak + i + 1))) ++i;
019      *(ak + j) = *(ak + i);
020      j = i;
021      i += k;
022      i = (30 * i) - 195 - (210 * (k / 15));
023      if(less(*(a + i), *(a + i + 15))) i += 15;

024      if(less(*(a + i), x)) {
025         if(less(*(ak + j), x)) {
026            if(less(*(ak + ((j - 3) >> 2)), x)) {
027               *(ak + ((j - 1) >> 1)) = *(ak + ((j - 3) >> 2));
028               if(less(*(ak), x)) {
029                  *(ak + ((j - 3) >> 2)) = *(ak);
030                  *(ak) = x;
031               }
032               else
033                  *(ak + ((j - 3) >> 2)) = x;
034            }
035            else
036               *(ak + ((j - 1) >> 1)) = x;
037         }
038         else
039            *(ak + j) = x;
040         return;
041      }
042      else
043         *(ak + j) = *(a + i);
```

```
044   }

045   if ((((240 * (i / 15)) + 15) > p.last_leaf()) && ((i + 14) <= p.last_leaf()))
{
046      position ak = a + i;
047      i = (less(*(ak + 1), *(ak + 2))) ?  2 :  1;
048      if(less(*(ak + i), x)) {
049         *(ak) = x;
050         return;
051      }
052      *(ak) = *(ak + i);
053      j = i;
054      i = (i << 1) + 1;
055      if(less(*(ak + i), *(ak + i + 1))) ++i;
056      if(less(*(ak + i), x)) {
057         *(ak + j) = x;
058         return;
059      }
060      *(ak + j) = *(ak + i);
061      j = i;
062      i = (i << 1) + 1;
063      if(less(*(ak + i), *(ak + i + 1))) ++i;
064      if(less(*(ak + i), x)) {
065         *(ak + j) = x;
066         return;
067      }
068      *(ak + j) = *(ak + i);
069      *(ak + i) = x;
070      return;
071   }

072   while (p.last_child(i) <= p.last_leaf()) {
073      j = p.top_all_present(a, i, less);
074      if (less(x, *(a + j))) {
075         p.update_all_present(a, i, *(a + j), less);
076      }
077      else {
078         p.update_all_present(a, i, x, less);
```

```
079         return;
080       }
081       i = j;
082    }
083    while(p.first_child(i) <= p.last_leaf()) {
084       if(p.last_child(i) <= p.last_leaf())
085          j = p.top_all_present(a, i, less);
086       else
087          j = p.top_some_absent(a, i, less);
088       if (less(x, *(a + j))) {
089          p.update_all_present(a, i, *(a + j), less);
090          i = j;
091       }
092       else {
093          p.update_all_present(a, i, x, less);
094          return;
095       }
096    }
097    p.update_all_present(a, i, x, less);
098    return;
099 }
```

A.3 Heap function code

```
001 template <typename position, typename ordering, typename policy>
002 void pop_heap(position a, const ordering& less, policy& p) {
003    typedef typename policy::index index;
004    typedef typename policy::element element;

005    index i = 0;
006    index j = p.last_leaf();
007    element x = *(a + j);
008    *(a + j) = *(a + i);
009    p.erase_last_leaf(a);
010    sift_down<d, position, index, element, ordering, policy>(a, i, x, less,
p);
```

```
011 }
```

Bibliography

[1] H. Aho and Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley (1974).

[2] M. D. Atkinson, J. R. Sack, N. Santoro, and T. Strothotte, Min-max heaps and generalized priority queues, *Communications of the ACM* **29** (1986), 996–1000.

[3] J. Bojesen, J. Katajainen, and M. Spork, Performance engineering case study: heap construction, *The ACM Journal of Experimental Algorithmics* **5** (2000), Article 15.

[4] S. Carlsson, The deap—A double-ended heap to implement double-ended priority queues, *Information Processing Letters* **26** (1987), 33–36.

[5] S. Carlsson, A variant of heapsort with almost optimal number of comparisons, *Information Processing Letters* **24(4)** (1987), 247–250.

[6] S. Carlsson, An optimal algorithm for deleting the root of a heap, *Information Processing Letters* **37** (1991), 117–120.

[7] S. Carlsson, J. Chen, and C. Mattsson, Heaps with bits, *Theoretical Computer Science* **164** (1996), 1–12.

[8] S. Carlsson, J. Chen, and T. Strothotte, A note on the construction of the data structure "deap", *Information Processing Letters* **31** (1989), 315–317.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, McGraw-Hill Higher Education (2001).

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

[11] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computations, *Communications of the ACM* **31** (1988), 1343–1354.

[12] S. Edelkamp and P. 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** (2002), Article 5.

[13] R. Floyd, Algorithm 245, treesort 3, *Communication of the ACM* **7(12)** (1964), 701.

[14] G. Franceschini and V. Geffert, An in-place sorting with $o(n \log n)$ comparisons and $o(n)$ moves, *Journal of the ACM* **52** (2005), 515–537.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, Cache-oblivious algorithms, *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society (1999), 285.

[16] G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM J. Comput.* **15** (1986), 964–971.

[17] G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM Journal of computing* **15** (1986), 964–971.

[18] T. Hagerup, Sorting and searching on the word RAM, *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **1373**, Springer-Verlag, Berlin/ Heidelberg (1998), 366–398.

[19] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers, Inc. (1996).

[20] ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission), *International Standard ISO/IEC 14882: Programming Languages — C++*, Genève (1998).

[21] B. 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`.

[22] C. Jensen and J. Katajainen, An experimental evaluation of navigation piles, CPH STL Report 2006-3, Department of Computing, University of Copenhagen (2006). Available at `http://www.cphstl.dk`.

[23] D. B. Johnson, Priority queues with update and finding minimum spanning trees, *Information Processing Letters* **4** (1975), 53–57.

[24] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer-Verlag, Berlin/Heidelberg (2001), 39–50.

[25] D. E. Knuth, *Sorting and Searching*, *The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman, Reading (1998).

[26] A. LaMarca and R. E. Ladner, The influence of caches on the performance of heaps, *The ACM Journal of Experimental Algorithmics* **1** (1996), Article 4.

[27] A. LaMarca and R. E. Ladner, The influence of caches on the performance of sorting, *Journal of Algorithms* **31** (1999), 66–104.

[28] C. McDiarmid and B. Reed, Building heaps fast, *Journal of Algorithms* **10** (1989), 352–365.

[29] J. I. Munro and V. Raman, Sorting with minimum data movement, *Journal of Algorithms* **13**,3 (1992), 374–393.

[30] J. I. Munro and V. Raman, Selection from read-only memory and sorting with minimum data movement, *Theoretical Computer Science* **165** (1996), 311–323.

[31] D. Ohashi, Cache oblivious data structures, Master's thesis, Department of Computer Science, University of Waterloo (2000).

[32] J. Pagter and T. Rauhe, Optimal time-space trade-offs for sorting, *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Los Alamitos (1998), 264–268.

[33] G. J. E. Rawlins, *Compared to What? An Introduction to the Analysis of Algorithms*, W. H. Freeman & Co., New York (1992).

[34] Silicon Graphics, Inc., *Standard Template Library Programmer's Guide* (1993–2003). Website accessible at `http://www.sgi.com/tech/stl/`.

[35] J. van Leeuwen and D. Wood, Interval heaps, *The Computer Journal* **36** (1993), 209–216.

[36] I. Wegener, The worst case complexity of McDiarmid and Reed's variant of Bottom-up Heapsort is less than $n \log n + 1.1n$, *Information and Computation* **97** (1992), 86–96.

[37] I. Wegener, Bottom-Up-Heapsort, a new variant of Heapsort beating, on an avarage, Quicksort (if n is not very small), *tcs* **118** (1993), 81–98.

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