Nordic Journal of Computing 10(2003), 238–262.

NAVIGATION PILES WITH APPLICATIONS TO SORTING, PRIORITY QUEUES, AND PRIORITY DEQUES

JYRKI KATAJAINEN
FABIO VITALE
Datalogisk Institut, Københavns Universitet
Universitetsparken 1, 2100 København Ø, Denmark
{jyrki|fabio}@diku.dk

Abstract. A data structure, named a navigation pile, is described and exploited in the implementation of a sorting algorithm, a priority queue, and a priority deque. When carrying out these tasks, a linear number of bits is used in addition to the elements manipulated, and extra space for a sublinear number of elements is allocated if the grow and shrink operations are to be supported. Our viewpoint is to allow little extra space, make a low number of element moves, and still keep the efficiency in the number of element comparisons and machine instructions. In spite of low memory consumption, the worst-case bounds for the number of element comparisons, element moves, and machine instructions are close to the absolute minimum.

ACM CCS Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems – *sorting and searching*; E.1 [Data Structures] – *lists, stacks, and queues*; E.2 [Data Storage Representations] – *linked representations*; D.3.2 [Programming Languages]: Language Classifications – *object-oriented languages*

Key words: algorithms, sorting, data structures, priority queues, priority deques, heaps

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 [6].

The STL, which is an integrated part of the ISO standard for the C++ programming language [12], 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 [13]. 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.

Received July 9, 2003; accepted November 22, 2003.

(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 [17, 18].

(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 [11]. 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, [12, §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 [25] and the generalization to $d > 2$ was suggested by Johnson [14].

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 [12, §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 [15] 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 [16].

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 2, and it is made dynamic in Section 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 $n \log_2 - n \log_2 e + (1/2) \log_2 n + \Theta(1)$ [16, §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 [19]. Pilesort is described and analysed in Section 3.

In Section 5 we show how navigation piles can be employed in the implementation of priority deques. In Section 6 we consider some generalizations of navigation piles. In Section 7 the results proved are summarized (see Table I) 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 [20], 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.* [4] 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 [22], 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 [15] — 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, [4, 8, 24]). 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. [4, 8, 24]) 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 [9] 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 [19] 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 [1], deaps [3] (see also [5]), and interval heaps [23]. All these structures can be made dynamic using the techniques described in [15], 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.

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 $2^{\eta+1}-1$.

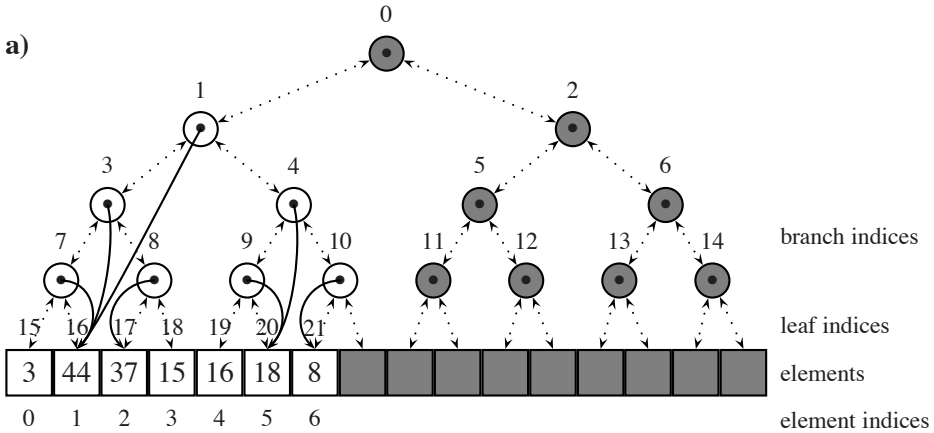
Leaves: If there are n elements, $n \leq 2^\eta$, the first n leaves store one element each and the remaining leaves are empty.

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. 1 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:



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	*	*	*	*

Fig. 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.

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, $0 \le \delta < \eta$. 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:

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

Requirement: $\gamma \le w$.

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: $\gamma \le w$.

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.* [2] 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^η be the smallest power of 2 larger than or equal to *capacity*;
Allocate space for the bit sequence $B[0\,..\,2^{\eta+1})$;
Use n as a synonym for $A.size()$;
if $(n \le 1)$ **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 $k \leftarrow \ell$; $k \ge j$; $--k$)
    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) $\leftarrow jump\_to\_element(second\_child(i))$;
    $B[s\,..\,s+\gamma) \leftarrow less(A[m], A[\ell])\ ?\ \lambda : 2^{\gamma-1}+\mu$;
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,·) ← *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, [10]), 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 ← *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 Δ ← *height*(*root*());
node_index j ← *last_leaf*();
while ($\Delta > 0$)
    level *half* ← $\lfloor \Delta/2 \rfloor$;
    branch_index i ← *ancestor*($j, \Delta - half$);
    (element_index $\ell, \cdot$) ← *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 $O(\log_2 n)$.

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. 2 on page 251). 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 $m, \cdot$) $\leftarrow$ *jump_to_element*(*root*());
const element_index $\ell \leftarrow$ *leaf_to_element*(*last_leaf*());
(branch_index i_1, branch_index i_2) $\leftarrow$
    first_two_ancestors_with_more_than_one_child(*last_leaf*());
(element_index $k, \cdot$) $\leftarrow$ *jump_to_element*(i_2);
element_index j_1;
if (*height*(i_1) $= 1$)
    $j_1 \leftarrow \ell - 1$;
else
    ($j_1, \cdot$) $\leftarrow$ *jump_to_element*(*first_child*(i_1));
if ($m = \ell$)
    A.*pop_back*();
    update_full_path(*last_leaf*());
elseif ($k \neq \ell$)
    $A[m] \leftarrow A[\ell]$;
    A.*pop_back*();
    update_partial_path(i_1, i_2, *element_to_leaf*(j_1), *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

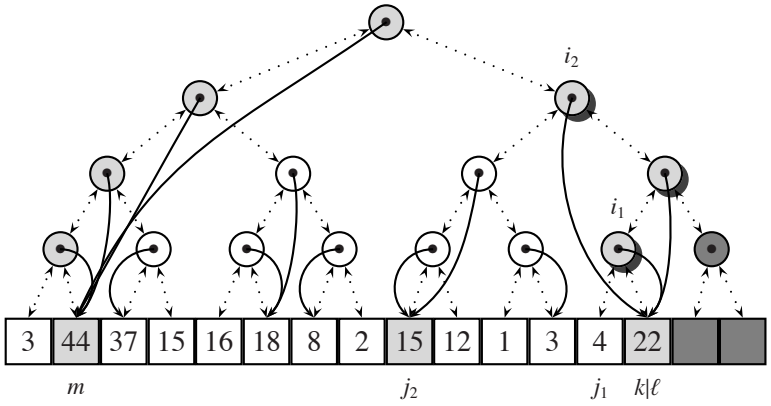


Fig. 2: Illustration of Case 3. The nodes whose contents may change are indicated in light gray. When updating the contents of the shadowed branches on the right, no element comparisons are necessary.

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. 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 [25]. 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 [12, §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, [24]), 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 $n \log_2 n + 0.59n + O(1)$.

4. Priority queues

In Section 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 [15].

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 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 [15] 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 2 can be used.

The data structure is illustrated in Fig. 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 [15]. 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 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 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.

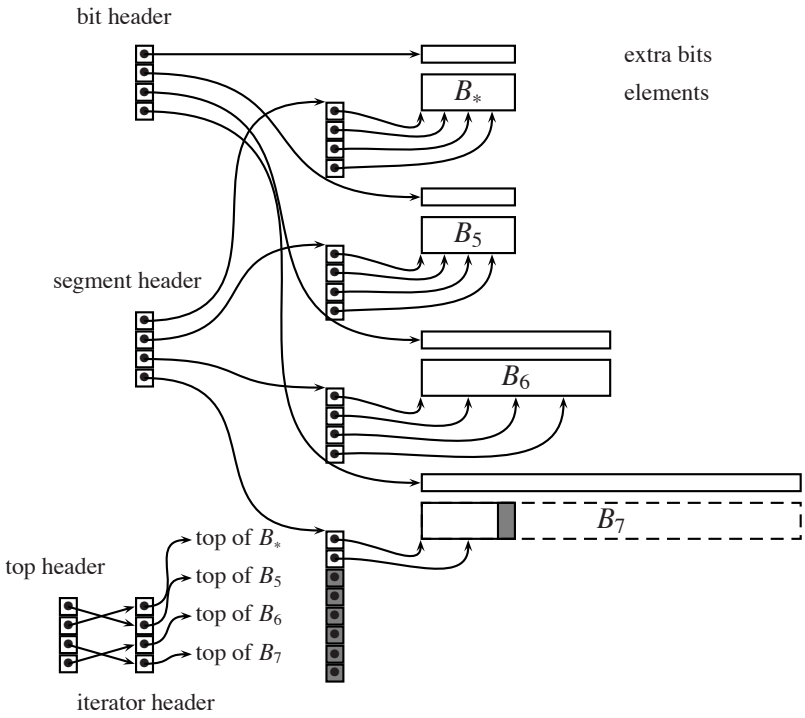


Fig. 3: A collection of navigation piles storing 77 elements. A space-efficient resizable array is used as the container for the elements. The gray areas denote empty memory segments allocated for dynamization purposes.

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.

5. Priority deques

To transfer our static and dynamic priority queues to priority deques, we use the twin technique described in [16, 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, [21, §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 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. 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.

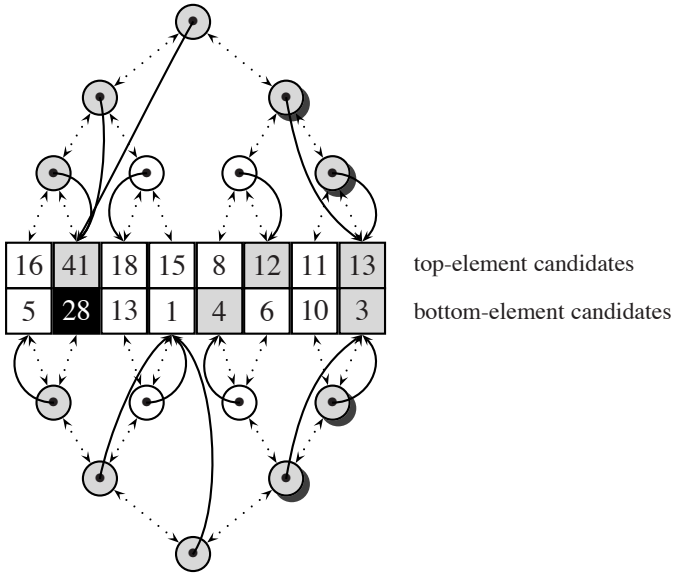


Fig. 4: Illustration of the *pop_top*() function. We assume that the last nonempty blocks are only of size 8, that the top element is inside the last nonempty block, and that $B_\#$ is empty. The vertical alignment of the leaves indicates the twin relationship. As in Fig. 2, the nodes whose contents 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 $B_\#$.

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 [20] 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.

7. Conclusions

Our results are summarized in Table I. 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 [7] 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 [12]. 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.

TABLE I: 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	$O(n)$	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$

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. 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.

Acknowledgements

We thank Jakob Krarup for bringing the authors together. Also, we thank the referees for their constructive comments.

References

[1] Atkinson, M. D., Sack, J. R., Santoro, N., and Strothotte, T. 1986. Min-max Heaps and Generalized Priority Queues. *Communications of the ACM 29*, 996–1000.

[2] Bojesen, J., Katajainen, J., and Spork, M. 2000. Performance Engineering Case Study: Heap Construction. *The ACM Journal of Experimental Algorithmics 5*, Article No. 15.

[3] Carlsson, S. 1987. The Deap—A Double-Ended Heap to Implement Double-Ended Priority Queues. *Information Processing Letters 26*, 33–36.

[4] Carlsson, S., Chen, J., and Mattsson, C. 1996. Heaps with Bits. *Theoretical Computer Science 164*, 1–12.

[5] Carlsson, S., Chen, J., and Strothotte, T. 1989. A Note on the Construction of the Data Structure "Deap". *Information Processing Letters 31*, 315–317.

[6] Department of Computing, University of Copenhagen. 2000–2003. *The CPH STL*. Website accessible at `http://www.cphstl.dk/`.

[7] Driscoll, J. R., Gabow, H. N., Shrairman, R., and Tarjan, R. E. 1988. Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computations. *Communications of the ACM 31*, 1343–1354.

[8] Edelkamp, S. and Stiegeler, P. 2002. Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ Comparisons. *The ACM Journal of Experimental Algorithmics 7*, Article No. 5.

[9] Franceschini, G. and Geffert, V. 2003. An In-Place Sorting with $O(n \log n)$ Comparisons and $O(n)$ Moves. *arXiv.org e-Print Archive*, Article No. 0305005. Available at `http://arxiv.org/abs/cs/`.

[10] Gonnet, G. H. and Munro, J. I. 1986. Heaps on Heaps. *SIAM Journal on Computing 15*, 964–971.

[11] Hagerup, T. 1998. Sorting and Searching on the Word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, Volume 1373 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg, 366–398.

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

[13] Jensen, B. S. 2001. *Priority Queue and Heap Functions*. CPH STL Report 2001-3, Department of Computing, University of Copenhagen, Copenhagen. Available at `http://www.cphstl.dk/`.

[14] Johnson, D. B. 1975. Priority Queues with Update and Finding Minimum Spanning Trees. *Information Processing Letters 4*, 53–57.

[15] Katajainen, J. and Mortensen, B. B. 2001. Experiences with the Design and Implementation of Space-Efficient Deques. In *Proceedings of the 5th Workshop on Algorithm Engineering*, Volume 2141 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg, 39–50.

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

[17] LaMarca, A. and Ladner, R. E. 1996. The Influence of Caches on the Performance of Heaps. *The ACM Journal of Experimental Algorithmics 1*, Article No. 4.
[18] LaMarca, A. and Ladner, R. E. 1999. The Influence of Caches on the Performance of Sorting. *Journal of Algorithms 31*, 66–104.
[19] Munro, J. I. and Raman, V. 1996. Selection from Read-Only Memory and Sorting with Minimum Data Movement. *Theoretical Computer Science 165*, 311–323.
[20] Pagter, J. and Rauhe, T. 1998. Optimal Time-Space Trade-Offs for Sorting. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Los Alamitos, 264–268.
[21] Rawlins, G. J. E. 1992. *Compared to What? An Introduction to the Analysis of Algorithms*. W. H. Freeman & Co., New York.
[22] Silicon Graphics, Inc. 1993–2003. *Standard Template Library Programmer's Guide*. Website accessible at `http://www.sgi.com/tech/stl/`.
[23] van Leeuwen, J. and Wood, D. 1993. Interval Heaps. *The Computer Journal 36*, 209–216.
[24] Wegener, I. 1992. 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*, 86–96.
[25] Williams, J. W. J. 1964. Algorithm 232: Heapsort. *Communications of the ACM 7*, 347–348.