An experimental evaluation of navigation piles[*]

Claus Jensen Jyrki Katajainen

Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark

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

1. Introduction

A *priority queue* is a data structure which stores a collection of elements and supports the operations *construct*, *push*, *pop*, and *top* in terminology used in the C++ standard [1, §23.2]. *top* retrieves the maximum element with respect to an ordering function given at the time of construction. For the realization of a priority queue, the navigation pile, introduced by Katajainen and Vitale [8], is an alternative to the standard binary heap [16]. In its basic form, a navigation pile is a static data structure where the maximum number of elements to be stored must be known in advance. The main advantage of navigation piles is that they provide fast worst-case priority-queue operations and have low space requirements (linear number of bits in addition to the elements stored). The same holds true even if the data structure is made fully dynamic.

At the beginning of this study we wanted to obtain a greater understanding of the practical utility of compact navigation piles and to study the practical utility of other adaptations of that data structure. In this paper we report the results of our experiments where we consider three different implementations of navigation piles. The first implementation, called a *compact pile*,

[*]Partially supported by the Danish Natural Science Research Council under contract 21-02-0501 (project "Practical data structures and algorithms").

CPH STL Report 2006-3. January 2006. Revised May 2006.

is based on the original description given in [8]. The elements are stored in a resizable array and above this array a bit container is built which stores navigation information in packed form. In the second implementation the bit container is replaced with an index array so we call it an *index pile*. In the third implementation the whole data structure is realized using concrete nodes and pointers; from now on we call it a *pointer-based pile*.

When analysing the runtime complexity of priority-queue operations and the space complexity of data structures, we use *word RAM* as our model of a computer (for a precise definition of the model, see [5]). In an *element comparison* the relative order of two elements is determined by evaluating the specified ordering function, which defines a strict weak ordering on the set of elements manipulated (for a definition of strict weak ordering, see, for example, [1, §25.3]). By an *element move* we mean the execution of a copy operation, copy construction or copy assignment, invoked for copying elements. We use the term *cost* to denote the sum of word-RAM instructions, element constructions, element destructions, and element comparisons performed.

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

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

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

Navigation piles in a nutshell. In order to define a navigation pile, which is a binary tree, we use the following standard terminology for trees. A node of a tree is the *root* if it has no parent, a *leaf* if it has no children, and a *branch* if it has at least one child. The *depth* of a node is the length of the path from that node to the root. The *height* of a node is the length of the

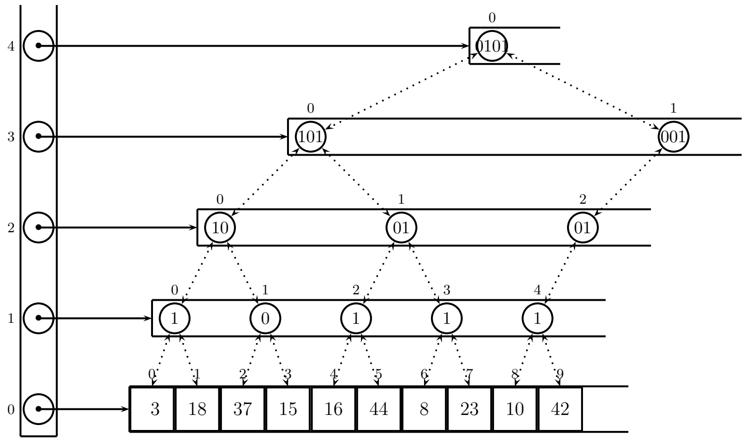

Figure 1. A dynamic compact pile storing 10 integers. For example, the bits $(001)_2$ in the second branch of height three indicate that the largest integer in the leaves of the subtree rooted at that branch is 42; the leaf has position 1 among the leaves spanned; indexing starts from 0. The offset of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

longest path starting from that node and ending at a leaf. In a *complete binary tree* all branches have two children and all leaves have the same depth.

A navigation pile can be seen as an extension of a selection tree described, for example, in [9]. In a *navigation pile* elements are stored at the leaves, and each branch stores a reference to the maximum element held in the leaves of the subtree rooted at that branch. The references at branches, referred to as *navigation information*, can be stored in different ways. According to the original proposal of Katajainen and Vitale [8], navigation information is stored in a packed form, but each branch could also store an index of the corresponding maximum element in the resizable array storing the elements.

Our implementation of a compact pile follows closely the guidelines given in [8]. An index pile is a natural extension where the packing/unpacking of references is omitted. Both data structures are static so the maximum capacity N must be known beforehand. A dynamic navigation pile, as described in [8], is a collection of static navigation piles. Therefore, the performance of our static implementations can be used as a baseline for the performance of the dynamic version as well.

The data-structural transformation described in [8] uses at most logarithmic number of static data structures to represent a dynamic data structure. This approach is theoretically sound, but tedious to implement. Therefore, we describe a new, simpler, and more direct way of dynamizing the data structure. The new strategy is to use one resizable array to store the branches of the same height and maintain a header which stores references to these resizable arrays (for an illustration, see Figure 1). That is, there can be at most $\lceil \log_2 n \rceil$ resizable arrays and the objects stored at each array are of the same size even if packing is used. The header itself is also a resizable array. A similar dynamization strategy has previously been used in connection with deques (see [7]). Even though this strategy requires some administrative work for maintaining the header and allocating/deallocating arrays "on the fly", the extra work should not increase the cost of the priority-queue operations significantly. The complexity of the operations used to navigate through a navigation pile, as for instance the calculation of the first and second child of a branch, is not increased and, therefore, these operations should not increase the cost of the priority-queue operations.

If N denotes the capacity of a static data structure and n the number of elements stored prior to each priority-queue operation, a compact pile and an index pile give the following performance guarantees [8]:

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

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

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

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

In addition to the space required for storing the elements, a pointer-based pile requires at most $5n + O(\log_2 n)$ words of extra space to store the pointers, which could be reduced to $4n + O(\log_2 n)$ words by using the child-sibling representation of binary trees (see, e.g. [15, Section 4.1]).

Experimental setting. As an immediate contender of our implementations of navigation piles, we chose the priority-queue implementation relying on implicit binary heaps available at the Free Software Foundation, Inc. implementation of the C++ standard library (shipped with `g++` compiler version 3.3.4). This heap implementation is known to be highly tuned;

from now on we refer to this as an *implicit heap.* Implicit binary heaps have also been used in many of the earlier experimental studies (see, for example, [6, 10, 11, 13]), which make it possible to compare our results indirectly to these earlier results.

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

A generic priority queue, as defined in the C++ standard [1], should be able to perform well under various circumstances. It should handle built-in types, user-defined types, and different types of ordering functions efficiently. In an attempt to cover the effects of a broad selection of possible input parameters with a reasonable number of experiments, we chose input parameters that represent a variation of cheap and expensive element comparisons and cheap and expensive element moves.

In the experiments the following types of input parameters were used: 1) built-in unsigned integers; 2) bigints, as described in the book of Bulka and Mayhew [2], which represent an unsigned integer as a string of digits; and 3) built-in unsigned integers combined with an ordering function that computes the natural logarithm of the elements before comparing them. In the case of built-in unsigned integers, both element comparisons and element moves are cheap. In the case of bigints, element comparisons are cheap, but the element moves are expensive. In the last case element comparisons are expensive, but the element moves are cheap. In earlier studies, similar settings for input parameters have been used. LaMarca and Landner [10, 11] used 32-bit and 64-bit integers and Sanders [13] used 32-bit integer keys with a 32-bit satellite data attached to the key. In experimental studies on sorting, elements consisting of 100-byte records with 10-byte keys are often used (see, e.g. [12]). Edelkamp and Stiegeler [4] used a number of expensive ordering functions, of which the expensive ordering function used by us is one.

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

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

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

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

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

These operation-generation models have also been used in several other experimental studies of priority queues (see [6, 10, 13] and the references therein).

2. Our implementations of navigation piles

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

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

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

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

In a compact pile, the nodes are also implicit, and there is the same interconnection between the leaves to elements stored in a resizable array as in an index pile. The navigation information held in branches is stored in a resizable bitarray. To compress the indices stored at branches as much as possible, for each branch we store a relative index of a leaf inside the leaf-subarray spanned by that branch. Using this relative index and the offset of the spanned subarray from the beginning of the element array, the actual position of the element referred to can be calculated (cf. Figure 1). For example, if a branch spans a group of four elements, the relative index can be stored as a number between zero and three which can be represented as a bit-pattern of size two. As a consequence of relative indexing the total amount of space needed for storing the navigation information can be brought down to $\lceil 2N/w \rceil$ words, N being the total capacity and w the size of a machine word. The utility functions for a compact pile are in many ways the same as those for an index pile; the main difference is the packing and unpacking of navigation information and the additional offset calculations required.

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

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

The maximum element to be returned by *top* is found by following the reference stored at the root.

When *pop* is executed, the maximum element referred to by the root is erased and the navigation information is updated accordingly. This update is done in three different ways depending on the circumstances inside the navigation pile (for an illustration of one of the cases, see Figure 2). The implementation details of the update depend on the form of the navigation pile. For a compact pile and an index pile, *pop* is accomplished in a similar

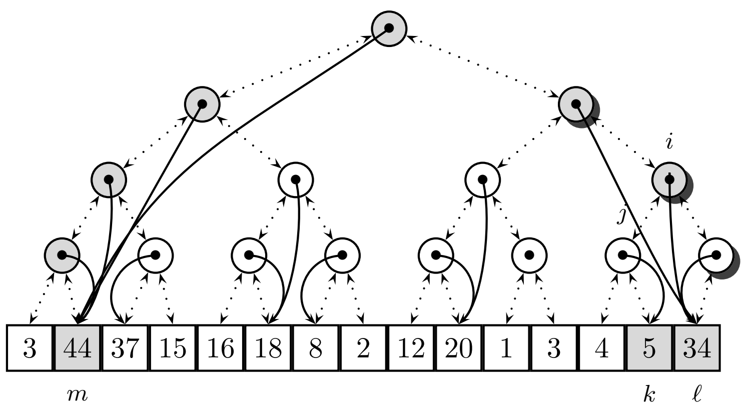

Figure 2. Illustration of the first-ancestor technique. The nodes, the contents of which may change, are indicated with light gray. When updating the contents of the shadowed branches on the right, no element comparisons are necessary.

manner as follows.

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

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

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

3. Experimental setup

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

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

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

We performed the experiments in the following environment:

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

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

All time measurements were done using the benchmark tool (Benz) developed by Katajainen and others (for a documentation, see [14]). Benz was configured to measure the CPU-time consumption of given operation sequences. In the framework of Benz, each experiment is repeated several times until 90% of the outcomes differ at most 20% from the median, which is reported; or more than 100 trials have been done after which the experiment is aborted.

4. Results

The results of our experiments for the sort model are given in Figures 3 (unsigned integers), 4 (bigints), and 5 (expensive comparisons). When manipulating integers the execution time for all variants of navigation piles is much higher than the execution time for implicit heaps, as shown by Figure

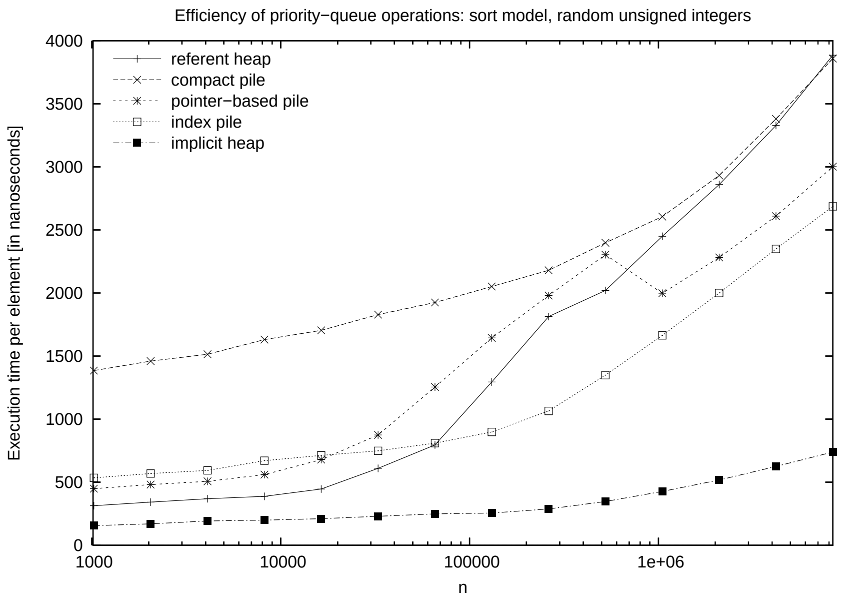

Figure 3. Performance of the implementations for the sort model using random unsigned integers on an Intel Pentium 4 workstation.

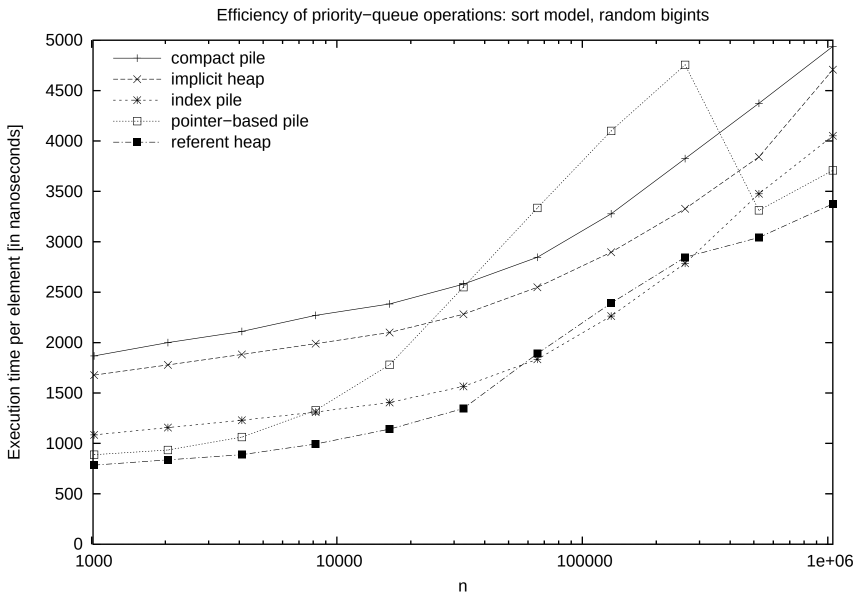

Figure 4. Performance of the implementations for the sort model using random bigints on an Intel Pentium 4 workstation.

3. The execution time for referent heaps is lower than the execution time for pointer-based piles when the number of elements is small, but after the

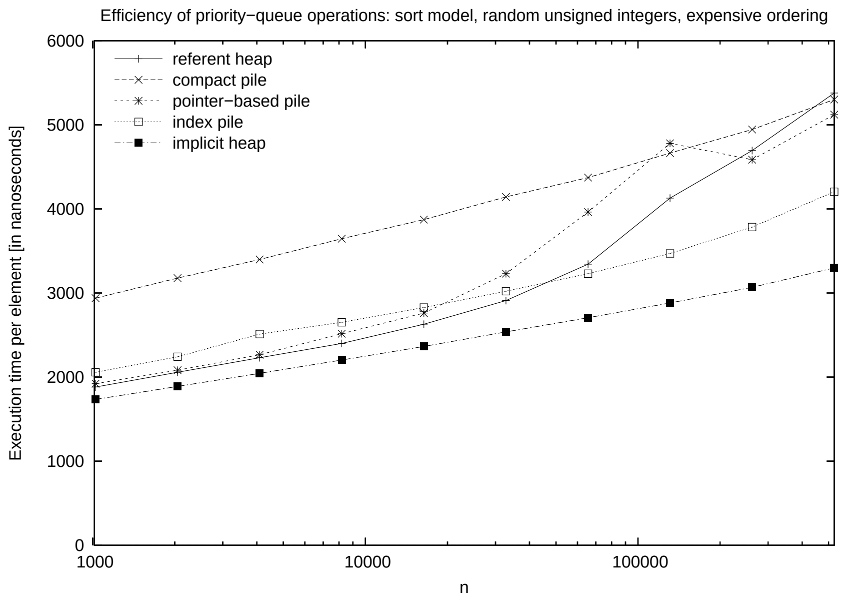

Figure 5. Performance of the implementations for the sort model using random unsigned integers with an expensive ordering function on an Intel Pentium 4 workstation.

crossover point (at about 500 000 elements) the execution time for referent heaps is higher. If referential integrity should be provided, pointer-based piles are a viable alternative to heaps even for integer input data. The tendency seen in the results for integer data is repeated in the experiments where an expensive ordering function is applied, but for large input sizes the differences in execution times decrease among the implementations. As seen in Figure 4, the advantages of navigation piles compared to implicit heaps become apparent when the elements considered are large and element moves become expensive.

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

5. Guidelines

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

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

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

Software availability

The programs used in this experimental study are accessible via the home page of the CPH STL project [3].

References

[1] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).

[2] D. Bulka and D. Mayhew, *Efficient C++: Performance Programming Techniques*, Addison Wesley Longman, Inc. (2000).

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

[4] 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.

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

[6] D. W. Jones, An emprirical comparison of priority-queue and event-set implementations, *Communications of the ACM* **29** (1986), 300–311.

[7] 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 (2001), 39–50.

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

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

[10] 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.

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

[12] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet, AlphaSort: A cache-sensitive parallel external sort, *The VLDB Journal* **4** (1995), 603–627.

[13] P. Sanders, Fast priority queues for cached memory, *The ACM Journal of Experimental Algorithmics* **5** (2000), Artilce 7.

[14] C. U. Søttrup and J. G. Pedersen, CPH STL's benchmark værktøj, CPH STL Report 2003-1, Department of Computing, University of Copenhagen (2003). Available at `http://www.cphstl.dk`.

[15] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM (1983).

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

Appendix

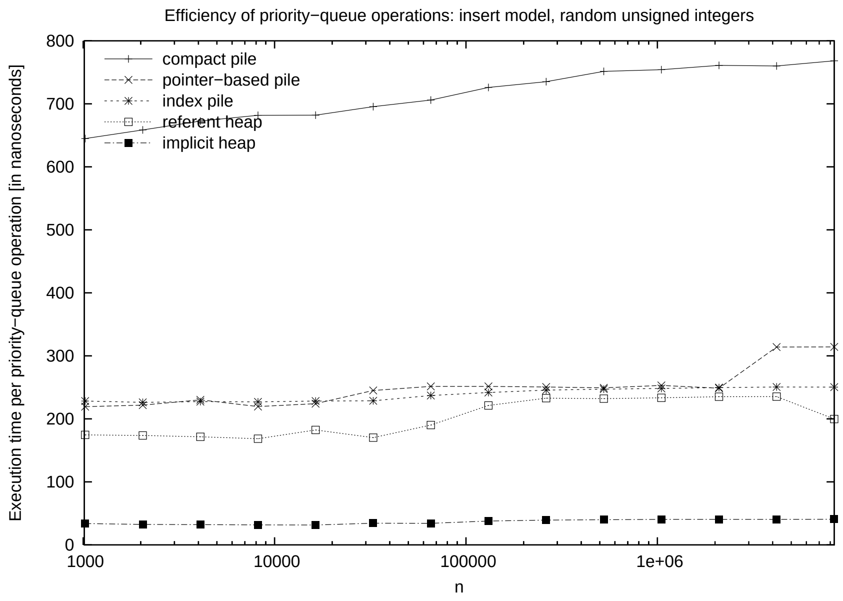

Figure 6. Performance of the implementations for the insert model using random unsigned integers on an Intel Pentium 4 workstation.

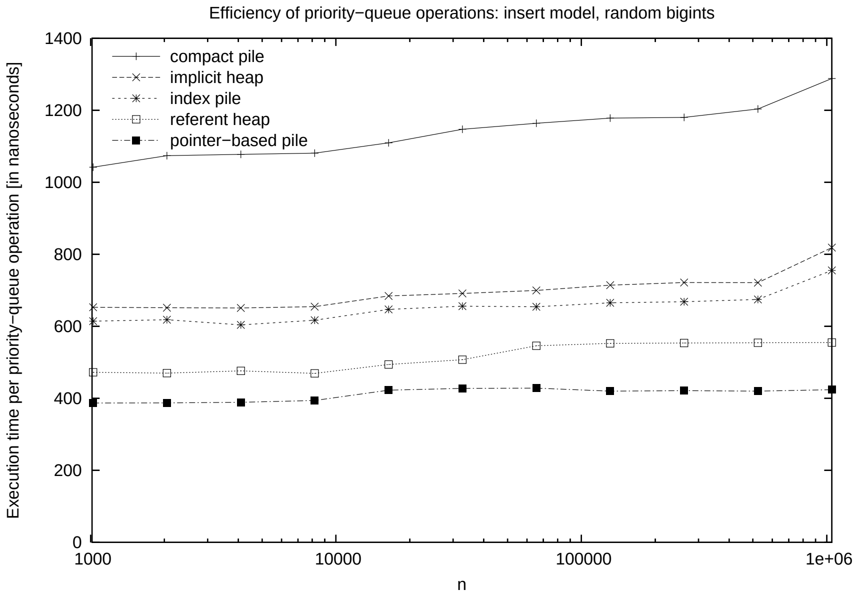

Figure 7. Performance of the implementations for the insert model using random bigints on an Intel Pentium 4 workstation.

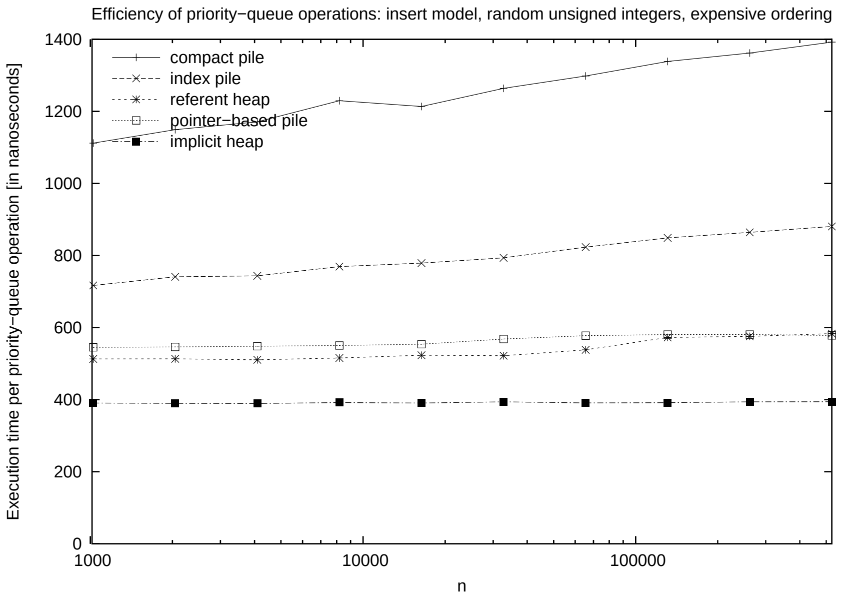

Figure 8. Performance of the implementations for the insert model using random unsigned integers with an expensive ordering function on an Intel Pentium 4 workstation.

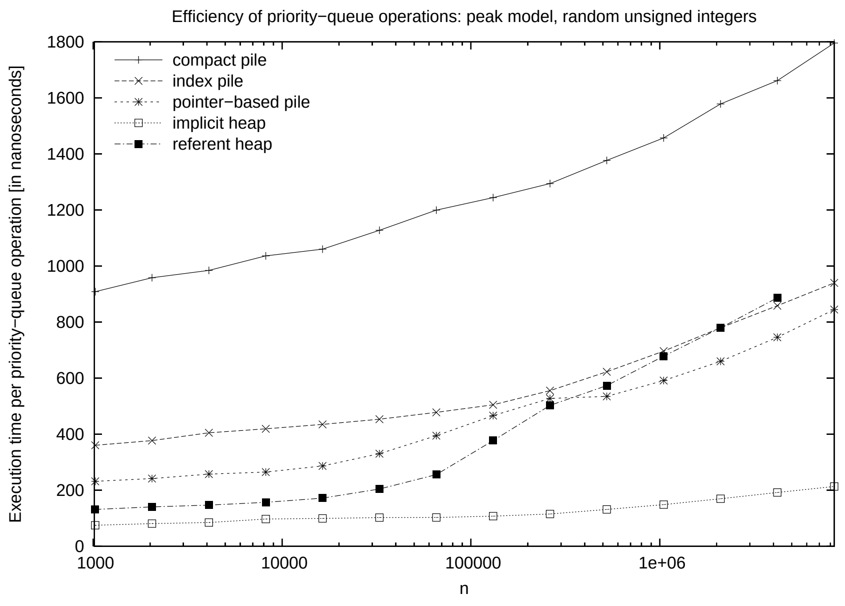

Figure 9. Performance of the implementations for the peak model using random unsigned integers on an Intel Pentium 4 workstation.

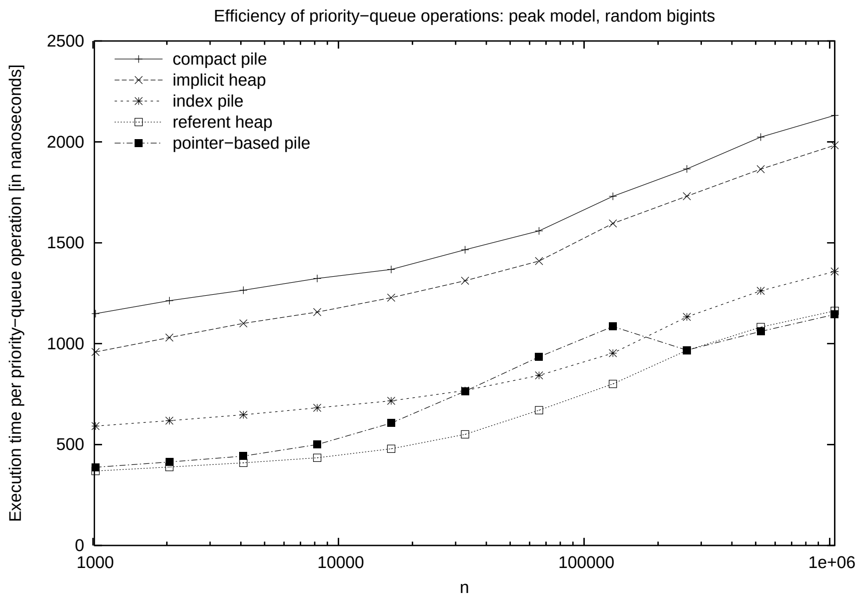

Figure 10. Performance of the implementations for the peak model using random bigints on an Intel Pentium 4 workstation.

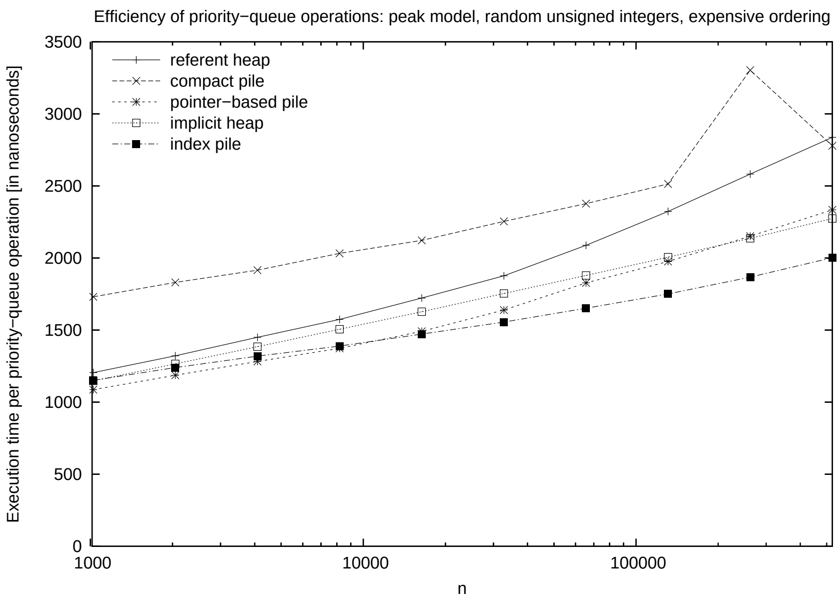

Figure 11. Performance of the implementations for the peak model using random unsigned integers with an expensive ordering function on an Intel Pentium 4 workstation.

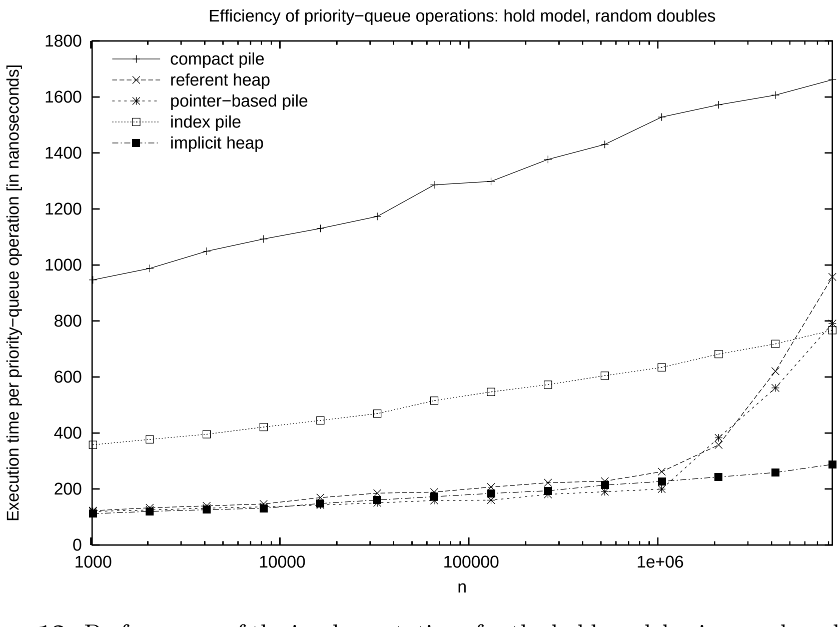

Figure 12. Performance of the implementations for the hold model using random doubles on an Intel Pentium 4 workstation.