

An extended truth about heaps^{*}

Claus Jensen¹, Jyrki Katajainen¹, and Fabio Vitale²

¹ Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
{surf, jyrki}@diku.dk

² Computer Science Department, University of Insubria
Via Ravasi 2, 21100 Varese, Italy
fabiovd@yahoo.com

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

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

1 Introduction

Context. The Standard Template Library (STL) is an integrated part of the standard library for the C++ programming language [6]. This work is part of the CPH STL project where the goal is to:

- study and analyse existing specifications for and implementations of the STL to determine the best approaches to optimization,
- provide an enhanced edition of the STL and make it freely available on the Internet,
- provide cross-platform benchmark results to give library users a better basis for assessing the quality of different STL components,
- develop software tools that can be used in the development of component libraries, and
- carry out experimental algorithmic research.

For further information about the project, see the CPH STL website [11].

^{*} CPH STL Report 2003-5, September 2003. Revised February 2004.

Goal of this study. Earlier studies by LaMarca and Ladner [22, 23] pointed out that the memory references are more local for 4-ary and 8-ary heaps than for binary heaps, which give better cache behaviour and faster programs. On the other hand, a later study by Sanders [28] showed that in some cases binary heaps are faster than 4-ary heaps. Therefore, the goal set for this study was

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

Preliminary definitions. For an integer $d \geq 2$, a *d-ary heap* — invented by Williams [33] for $d = 2$ and generalized for $d > 2$ by Johnson [17] — with respect to a given ordering function *less()* is a sequence of elements with the following properties:

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

Capacity: Each node of that tree stores one element.

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

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

Informally, such a heap is called a d -ary *max-heap*. We assume that the reader is familiar with the basic concepts related to heaps as described, for example, in [10, chap. 6].

An *iterator* is a generalization of a pointer that indicates a position of an element, provides operations for accessing the elements stored and operations for moving to neighbouring positions. All the iterators considered in this paper are assumed to be random-access iterators (for a precise definition of this concept, see [6, §24.1]). For example, if A and Z are random-access iterators and i is a nonnegative integer, $Z-A$, $++A$, $--Z$, $A+i$, $Z-i$, $A[i]$, and $*A$ are all allowable expressions having the same meaning as the corresponding pointer expressions. For iterators A and Z , we use $[A..Z)$ to denote the *sequence of positions* $A, A+1, \dots, A+((Z-A)-1)$ storing the elements $A[0], A[1], \dots, A[(Z-A)-1]$.

According to the C++ standard [6, §25.3.6] every realization of the standard library must provide the following *heap functions*:

template <typename position, typename ordering>

void *push_heap*(position A , position Z , ordering *less*);

Requirement: $[A..Z-1)$ stores a heap with respect to ordering function *less*().

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

template <typename position, typename ordering>

void *pop_heap*(position A , position Z , ordering *less*);

Requirement: $[A..Z)$ stores a heap with respect to ordering function *less*().

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

template <typename position, typename ordering>

void *make_heap*(position A , position Z , ordering *less*);

Effect: Convert the sequence stored in $[A..Z)$ into a heap with respect to ordering function *less*().

template <typename position, typename ordering>

void *sort_heap*(position A , position Z , ordering *less*);

Requirement: $[A..Z)$ stores a heap with respect to ordering function *less*().

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

When these functions are available, it is easy to implement a priority queue class (see [6, §23.2.3.2]) relying on any sequence supporting expansion and shrinkage at the back end.

Our work versus earlier work. The C++ standard [6, §25.3.6] gives tight complexity requirements for the heap functions. Letting n denote the size of the sequence manipulated, *push_heap*() should run in $O(\log_2 n)$ time and perform at most $\log_2 n$ element comparisons, *pop_heap*() should run in $O(\log_2 n)$ time and perform at most $2 \log_2 n$ element comparisons, *make_heap*() should perform at most $3n$ element comparisons, and *sort_heap*() at most $n \log_2 n$ element comparisons. Moreover, the intention is that the heap functions are memoryless. That is, no extra information is passed from one invocation to another, but temporary storage within a single function can be used.

These requirements were the main reason why we had to reject many of the proposals presented in the literature. Some methods only guarantee good amortized bounds, e.g. sequence heaps of Sanders [28]; some methods have too large constant factors in their worst-case complexity bounds, e.g. external heaps of Wegner and Teuhola [32]; and some methods require extra space, e.g. navigation piles of Katajainen and Vitale [20]. Also, in the current implementation of the CPH STL, done by Jensen [16], the heap functions based on 8-ary heaps do not fulfil the requirements, and in our preliminary experiments they turned out to be slow if element comparisons are expensive.

As far as we know, the heap functions based on 2-ary, 3-ary, and 4-ary heaps are the only ones that fulfil the requirements laid down in the C++ standard, except that for the complexity of the *sort_heap*() function. We describe a number of alternative implementations for the heap functions and provide a framework for benchmarking their performance. We apply the policy-based approach advocated by Alexandrescu [1, 2], as well as borrow elements from the earlier work done in our research group [4, 5, 16].

In the recent experimental studies on the efficiency of priority queues, the elements used as input were relatively small. LaMarca and Ladner [22, 23] used 32-bit and 64-bit integers, and Sanders [28] 32-bit integer keys associated with 32-bit satellite data. The focus in these studies was on cache behaviour and, when the elements manipulated are small, one can see dramatic cache effects. On the other hand, in the experimental studies on sorting it is a tradition to consider

larger elements: in industry-strength benchmarks 100-byte records with 10-byte keys are used (see, e.g. [25]). In the experiments of Edelkamp and Stiegeler [12] expensive ordering functions were considered.

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

1. Built-in unsigned ints, for which both element comparisons and element moves are cheap.
2. Bigints that represent an unsigned integer as a string of its digits (chars). The bigint class used by us is described in the book by Bulka and Mayhew [7, chap. 12]. For bigints element comparisons are relatively cheap, whereas element moves are expensive.
3. Unsigned ints with an ordering function that computes the natural logarithm of the given numbers before comparing them. This is just one of the expensive ordering functions considered by Edelkamp and Stiegeler [12].
4. Pairs of unsigned ints and bigints applying the natural logarithm function for unsigned ints in element comparisons. This makes both element comparisons and element moves expensive.

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

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

2 Implementation alternatives

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

According to the original proposals by Williams [33] and Johnson [17] the insertion is done by a simple linear scan starting from the new last leaf. One could reduce the number of element comparisons — keeping the number of element moves unchanged — by performing the test whether to stop the traversal only at every second level and backtrack if one goes too far up. Also, since the elements stored on the siftup path are in sorted order, binary search could be used to determine the final destination of the new element [8, 14]. Yet another alternative, as communicated to us by Pasanen [27], is to use exponential binary search [3] instead of binary search.

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

approach	reference	# comparisons	# moves
bottom-up			
– basic	[17, 33]	h	$h+2$
– two levels at a time	folklore	$\lfloor h/2 \rfloor + 1$	$h+2$
– binary search	[8, 14]	$\lceil \log_2(h+1) \rceil$	$h+2$
– exponential binary search	[27]	$2 \lfloor \log_2 h \rfloor + 1$	$h+2$

In Table 1 we summarize the worst-case performance of the implementation alternatives mentioned.

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

In the *top-down heapifying*, used in the original articles by Williams [33] and Johnson [17], the siftdown path is traversed down as long as the node under consideration is not a leaf and the element stored at that node is larger than the element put aside. Simultaneously, the elements met are moved one level up to get the hole down. When the traversal stops, the element put aside is stored at the hole. If the arity of the tree is d , at most d element comparisons are done at each level: at most $d-1$ to find the maximum element stored at the children and one to determine whether we should stop or not.

We could reduce the number of element comparisons by carrying out the test whether we should stop only at every second level, and backtracking if we proceed too far down. Another possibility, proposed by Gu and Zhu [15], is to calculate the height of the tree in the beginning, do the stop test first at level $\lfloor h/2 \rfloor$, and backtrack with linear search or repeat the process recursively downwards. That is, a one-sided binary search is performed on the sequence stored on the siftdown path. As shown in [9], it is possible to reduce the number of element comparisons even further, but this improvement is only of theoretical interest.

In the *bottom-up heapifying*, proposed by Floyd (as cited in [21, §5.2.3, exercise 18]), the siftdown path is first traversed down until a leaf is met, the elements on the path are moved one level up, and thereafter another traversal up the tree along the siftdown path is done to determine the final destination of the element put aside, by moving the elements met along. When traversing down at most $d-1$ element comparisons are done at each level, and when traversing up one element comparison is done at each level considered. The point is that often

Table 2. The worst-case performance of various versions of *pop_heap()*. If the heap stores $n \geq 1$ elements, the height h is equal to $\lceil \log_d((d-1)(n-1)+1) \rceil$.

approach	reference	# comparisons	# moves
top-down			
– basic	[17, 26, 33]	dh	$h+2$
– two levels at a time	folklore	$(d-1)h + \lfloor h/2 \rfloor + 1$	$h+2$
– one-sided binary search	[15]	$(d-1)h + \lceil \log_2(h+1) \rceil$	$h+2$
bottom-up			
– basic	[21, §5.2.3, exercise 18]	dh	$2h+2$
– two levels at a time	folklore	$(d-1)h + \lfloor h/2 \rfloor + 1$	$2h+2$
– binary search	[8, 14]	$(d-1)h + \lceil \log_2(h+1) \rceil$	$2h+2$
– exponential binary search	[27]	$(d-1)h + 2 \lfloor \log_2 h \rfloor + 1$	$2h+2$
– move saving	[31]	dh	$h+2$

only a few nodes are visited during the upwards traversal. As to the number of element comparisons, the worst case is the same as that for the basic top-down version, but the number of element moves can be almost twice as high.

The number of element comparisons performed in the worst case can be reduced in several ways depending on how the upwards traversal is done. First, the test whether to stop or not could be done only at every second level. Second, the final destination of the element put aside could be determined using binary search [8, 14]. Third, the final destination could be determined using exponential binary search [27]. Fourth, one could save element moves by omitting the moves during the downwards traversal and doing them first after the final destination of the new element is known. In the original proposal by Wegener [31] the upwards traversal was done by linear search, but any of the above-mentioned approaches could be used.

The worst-case performance of all the implementation alternatives mentioned is summed up in Table 2.

Function *make_heap()*. In the paper by Bojesen et al. [5] various heap construction methods were rigorously analysed both theoretically and experimentally. The conclusion was that the standard heap construction method [13] performs almost optimally in all respects if the nodes are handled in depth-first order. Our experiments show that the extreme code-tuning described in [5] is not necessary since on contemporary computers the few instructions saved can be executed in pipeline or even in parallel with other interdependent instructions. According to Okoma [26], for a d -ary heap containing n elements, the heap construction requires at most $dn/(d-1)$ element comparisons.

Function *sort_heap()*. It is well-known that heapsort cannot compete against quicksort or mergesort. Moreover, since quicksort does not have a good worst-case performance guarantee, we rely on mergesort in the realization of this function. Even if temporary storage could be allocated, we decided to avoid that — to make the function more robust — and use the in-place mergesort algorithm developed by Katajainen et al. [19].

```

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

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

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

private:
    index n;
};

```

Fig. 1. Declaration of the heap-policy class.

It is interesting to note that in the experiments reported in the original paper by Katajainen et al. [19] heapsort was a clear winner compared to in-place mergesort. Contemporary computers favour local memory accesses and most memory accesses in in-place mergesort are such, which makes it a noteworthy alternative. Experiments carried out in our research group by Sloth et al. [30] showed that for uniformly distributed integer data in-place mergesort is almost as fast as introsort [24], which is an adaptation of median-of-three quicksort. We want to point out that, even its practical utility, the implemented variant of in-place mergesort requires $n \log_2 n + O(n)$ element comparisons, so some fine-tuning is necessary to reach the bound $n \log_2 n$ set down in the C++ standard.

3 Benchmarking framework

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

It is natural to represent a policy as a class which contains all the core functions. The declaration of such a class is given in Fig. 1. Most member functions are **const** functions since they do not change the state of a policy object. The

only information which is stored in a policy object is n , the current number of elements in the heap.

The meaning of most member functions should be clear from their name, except that of the functions *top_all_present()* and *top_some_absent()* which compute for a given node the index of the child storing the maximum element among all the children. Internally, the two functions are similar — a linear scan over the children is performed and the index of the node storing the maximum element is recalled. To make an efficient loop unrolling possible, we separated the case where the branch considered has all its d children, and the case where some of the children is missing.

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

$$\begin{aligned} \text{depth}(i): & \lfloor \log_d((d-1)i+1) \rfloor \\ \text{leftmost_node}(\lambda): & (d^\lambda - 1)/(d-1) \\ \text{ancestor}(i, \lambda, \Delta): & \lfloor (i - \text{leftmost_node}(\lambda)) / d^\Delta \rfloor + \text{leftmost_node}(\lambda - \Delta), \text{ where } \Delta \in \\ & \{0, 1, \dots, \lambda\}. \end{aligned}$$

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

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

template <typename position, typename index, typename element, typename ordering, typename policy>

void *siftup*(position A , index i , element x , ordering *less*, policy& p);

Requirement: $A[i]$ contains a hole.

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

template <typename position, typename index, typename element, typename ordering, typename policy>

void *siftdown*(position A , index i , element x , ordering *less*, policy& p);

Requirement: $A[i]$ contains a hole.

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

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

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

4 Benchmarks

Overall strategy. We will not repeat the experiments for *make_heap()* done in [5], and those for *sort_heap()* (in-place mergesort) done in [30]. Therefore, our focus is on the functions *push_heap()* and *pop_heap()*. To compare the efficiency of the implementation alternatives presented, we decided to use artificial operation generation in our experiments. Also, we restrict the problem sizes so that the heaps can be in internal memory all the time.

In earlier experimental studies on the efficiency of priority queues (see [18, 22, 28] and the references therein), several different operation-generation models were used. To describe the models we found interesting, we use I as a short-hand notation for *push_heap()*, D for *pop_heap()*, and M_n for *make_heap()* involving n elements. For nonnegative integers k and n , the models considered by us were:

- Insert I^n :** Determine the average running time of a single *push_heap()* when n operations are executed in all.
- Hold $I^n(DI)^k$:** Determine the average running time for a single iteration (DI) after inserting n elements.
- Sort $M_n D^n$:** Determine the average running time per element when sorting a sequence of n elements.
- Peak $(IDI)^n(DID)^n$:** Determine the average running time per operation.

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

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

As explained in the introduction, we decided to use four kinds of input: unsigned ints; bigints [7, chap. 12]; unsigned ints with an expensive ordering function; and pairs containing unsigned ints as keys and bigints as an extra load, comparing the keys with an expensive ordering function. Uniformly generated random unsigned ints (32 bits) and bigints (strings of about 10 digits) were used all over. To give an idea of the execution times of various primitive operations, we accessed a vector containing unsigned ints or bigints both sequentially (stride $p = 1$) and arbitrarily (stride $p = 617$). The results of these micro benchmarks are shown in Table 3.

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

Table 3. Cost of some instructions on an Intel Pentium 4 workstation for **a)** unsigned ints and **b)** bigints; n denotes the size of the sequence accessed and p the stride used.

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

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

The experiments were carried out using the benchmark tool developed for the CPH STL by Katajainen and others [11]. This tool is written in Python; most other scripting was also done in Python. The tool was used to generate test drivers which measure the CPU time spent by given operation sequences. Each experiment was repeated several times depending on the clock precision and the median of the execution times was reported if the execution times of 90% of the runs were within 20% of the reported value; otherwise, the whole experiment was ignored. Based on the scripts created it should be relatively easy for others to repeat our experiments in other environments.

Results. In order to understand the conflicting results reported by LaMarca and Ladner [22] and Sanders [28], we analysed the test arrangements and found the following differences:

	LaMarca and Ladner [22]	Sanders [28]
approach	top-down	bottom-up
element type	64-bit integers	32-bit interger keys, 32-bit satellite data
operation generation	hold model	peak model
element generation	earlier minimum plus a random increment	random over the whole range
computer	Pentium and others	Pentium II and others

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

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

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

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

We carried out the final experiments in two rounds. In the first round, for each implementation alternative and for each kind of input we determined the best arity. If an alternative was the fastest for more than 80% of the data points measured, we declared it as a *winner*. In the second round, for each kind of input the winners were compared against each others. The results of these tests are shown for the four types of input in Figs. 2, 3, 4, and 5, respectively. In all experiments we used the functions *partial_sort()* (carefully coded heapsort using binary heaps and bottom-up heapifying) and *sort()* (introsort) from the Silicon Graphics Inc. implementation of the STL [29] as our reference implementations.

Discussion. When comparing the execution times of our implementations and the reference implementations, our framework is seen to have a clear abstraction overhead. Especially, this is true for integer data and small problem sizes. On the other hand, in some cases our programs are faster than the reference implementations. For example, when element comparisons are expensive and they dominate the overall running time, most heapsort variants are faster than introsort indicating that heapsort performs fewer element comparisons than quicksort. We hoped that the programs produced with our framework were faster. Now separate code tuning is necessary before the programs can be imported into the CPH STL.

For integer data our results are similar to those reported by LaMarca and Ladner [22, 23] even though on our Pentium 4 computer the point when the 4-ary heaps become faster than 2-ary heaps appears first at the end of the range plotted in Fig. 2. In all fairness other data types have to be considered, too. Our results show that a single arity is not the best for all kinds of elements and all kinds of ordering functions. Furthermore, no heapifying approach is the best for all kinds of elements and all kinds of ordering functions, but for randomly generated inputs it is very difficult to beat the basic bottom-up approach.

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

Software availability

All source code and scripts developed in the course of this study are available at the CVS repository for the CPH STL project [11].

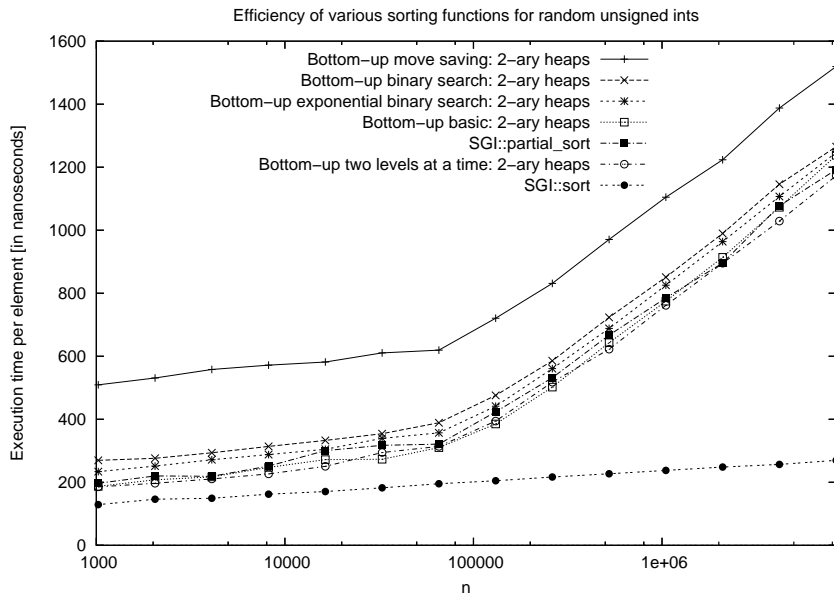


Fig. 2. Performance of the first-round winners for random unsigned ints on an Intel Pentium 4 workstation.

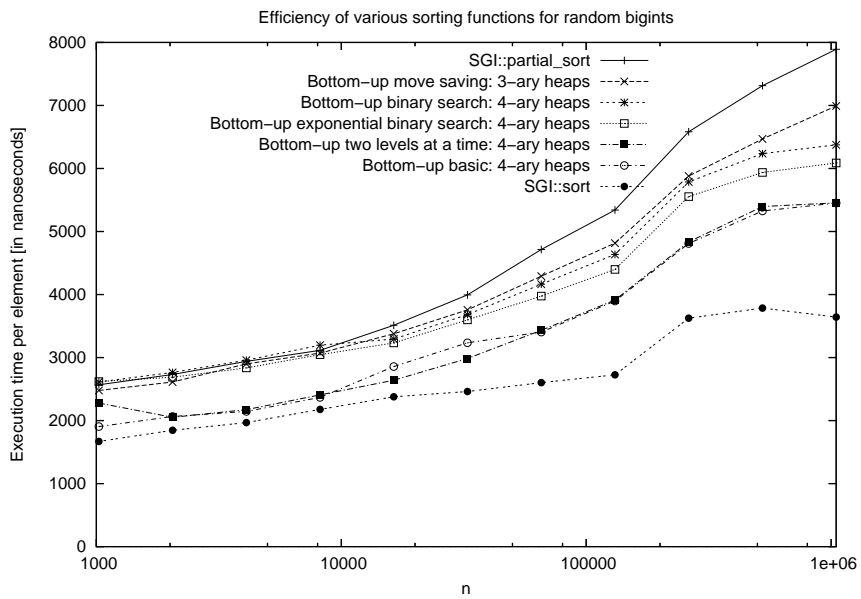


Fig. 3. Performance of the first-round winners for random bigints on an Intel Pentium 4 workstation.

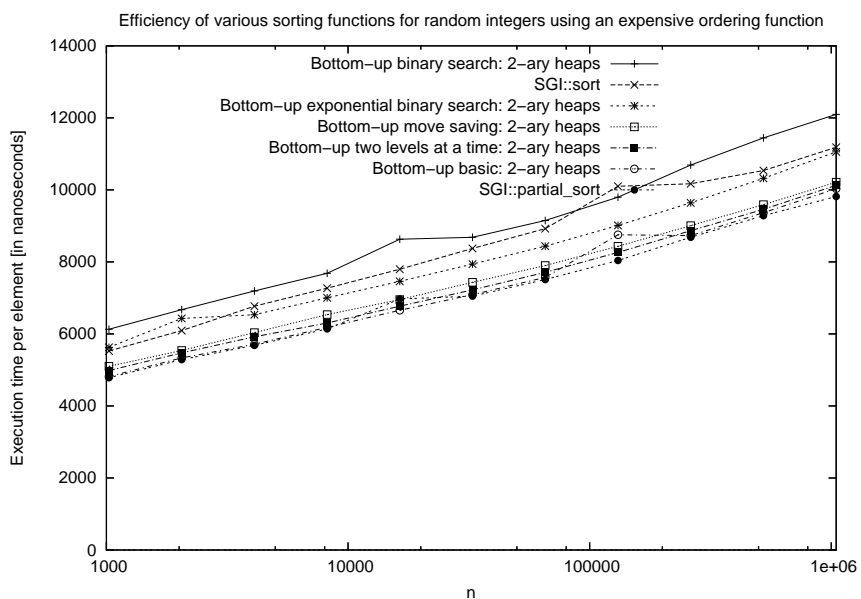


Fig. 4. Performance of the first-round winners for random unsigned ints with an expensive ordering function on an Intel Pentium 4 workstation.

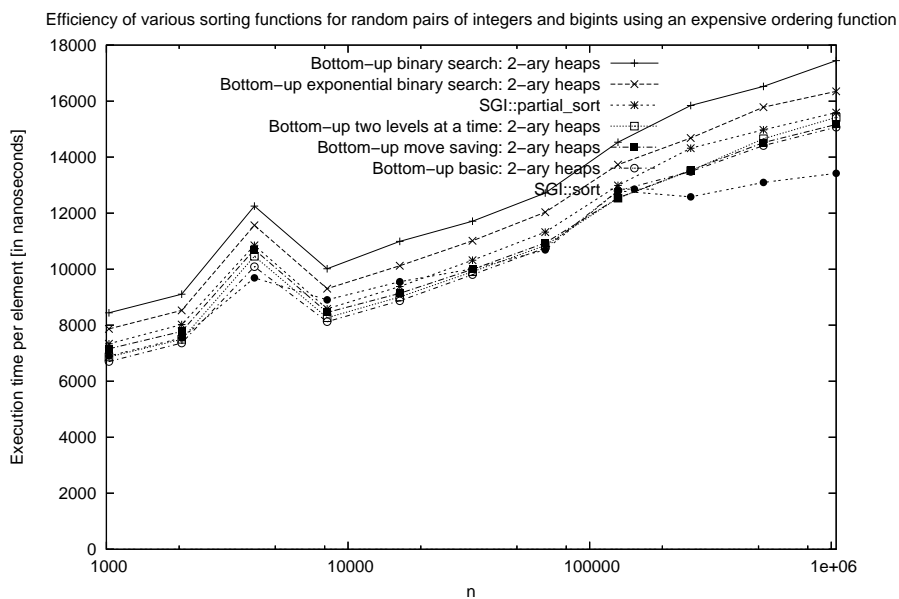


Fig. 5. Performance of the first-round winners for random pairs of unsigned ints and bigints with an expensive ordering function on an Intel Pentium 4 workstation.

Acknowledgements

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

References

- [1] A. Alexandrescu, Generic<programming>: A policy-based `basic_string` implementation, *C++ Expert Forum* (2001). Available at <http://www.cuj.com/experts/1096/toc.htm>.
- [2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Pattern Applied*, Addison-Wesley, Upper Saddle River (2001).
- [3] J.L. Bentley and A.C.C. Yao, An almost optimal algorithm for unbounded searching, *Information Processing Letters* **5** (1976), 82–87.
- [4] J. Bojesen, Heap implementations and variations, Written project, Department of Computing, University of Copenhagen, Copenhagen (1998). Available at <http://www.diku.dk/forskning/performance-engineering/Perfeng/resources.html>.
- [5] J. Bojesen, J. Katajainen, and M. Spork, Performance engineering case study: heap construction, *The ACM Journal of Experimental Algorithmics* **5** (2000), Article 15.
- [6] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).
- [7] D. Bulka and D. Mayhew, *Efficient C++: Performance Programming Techniques*, Addison-Wesley, Reading (2000).
- [8] S. Carlsson, A variant of Heapsort with almost optimal number of comparisons, *Information Processing Letters* **24** (1987), 247–250.
- [9] S. Carlsson, A note on Heapsort, *The Computer Journal* **35** (1992), 410–411.
- [10] T. H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press, Cambridge (2001).
- [11] Department of Computing, University of Copenhagen, The CPH STL, Website accessible at <http://www.cphstl.dk/> (2000–2004).
- [12] S. Edelkamp and P. Stiegeler, Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ comparisons, *The ACM Journal of Experimental Algorithmics* **7** (2002), Article 5.
- [13] R. W. Floyd, Algorithm 245: Treesort 3, *Communications of the ACM* **7** (1964), 701.
- [14] G.H. Gonnet and J.I. Munro, Heaps on heaps, *SIAM J. Comput.* **15** (1986), 964–971.
- [15] X. Gu and Y. Zhu, Optimal heapsort algorithm, *Theoretical Computer Science* **163** (1996), 239–243.
- [16] B. S. Jensen, Priority queue and heap functions, CPH STL Report 2001-3, Department of Computing, University of Copenhagen, Copenhagen (2001). Available at <http://www.cphstl.dk/>.
- [17] D. B. Johnson, Priority queues with update and finding minimum spanning trees, *Information Processing Letters* **4** (1975), 53–57.
- [18] D. W. Jones, An empirical comparison of priority-queue and event-set implementations, *Communications of the ACM* **29** (1986), 300–311.
- [19] J. Katajainen, T. Pasanen, and J. Teuhola, Practical in-place mergesort, *Nordic Journal of Computing* **3** (1996), 27–40.

- [20] J. Katajainen and F. Vitale, Navigation piles with applications to sorting, priority queues, and priority dequeues, *Nordic Journal of Computing* **10** (2003), 238–262.
- [21] D. E. Knuth, *Sorting and Searching, The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman, Reading (1998).
- [22] 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.
- [23] A. LaMarca and R. E. Ladner, The influence of caches on the performance of sorting, *Journal of Algorithms* **31** (1999), 66–104.
- [24] D. R. Musser, Introspective sorting and selection algorithms, *Software—Practice and Experience* **27** (1997), 983–993.
- [25] 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.
- [26] S. Okoma, Generalized Heapsort, *Proceedings of the 9th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science* **88**, Springer-Verlag, Berlin/Heidelberg (1980), 439–451.
- [27] T. A. Pasanen, Public communication, December 2003.
- [28] P. Sanders, Fast priority queues for cached memory, *The ACM Journal of Experimental Algorithmics* **5** (2000), Article 7.
- [29] Silicon Graphics, Inc., Standard template library programmer’s guide, Website accessible at <http://www.sgi.com/tech/stl/> (1993–2004).
- [30] J. Sloth, M. Lemvig, and M. Kristensen, Sorting i cph stl, CPH STL Report 2003-2, Department of Computing, University of Copenhagen, Copenhagen (2003). Available at <http://www.cphstl.dk/>.
- [31] I. Wegener, Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small), *Theoretical Computer Science* **118** (1993), 81–98.
- [32] L. M. Wegner and J. I. Teuhola, The external heapsort, *IEEE Transactions on Software Engineering* **15** (1989), 917–925.
- [33] J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7** (1964), 347–348.