Experimental evaluation of local heaps

Claus Jensen[1,*] Jyrki Katajainen[1,*] Fabio Vitale[2,†]

[1] *Department of Computing, University of Copenhagen*
Universitetsparken 1, 2100 Copenhagen East, Denmark
[2] *Department of Computing, University of Varese*
Via Ravasi 2, 21100 Varese, Italy

Abstract. In this paper we present a cache-aware realization of a priority queue, named a local heap, which is a slight modification of a standard binary heap. The new data structure is cache efficient, has a small computational overhead, and achieves a good worst-case performance with respect to the number of element comparisons and the number of element moves. We show both theoretically and experimentally that the data structure is competitive with a standard binary heap, provided that the number of elements stored is not small.

1. Introduction

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

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

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

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

A d***-ary heap***, which is an efficient way to implement a priority queue, is a left-complete d-ary tree, in which each node stores one element and which is represented in a sequence by storing elements in breadth-first order A d-ary heap, with respect to an ordering function *less*(), has the following crucial property: for each branch of the tree, the element y stored at that node is no smaller than the element x stored at any child of that node, i.e. *less*(y, x) must return false. The binary heaps were invented by Williams [11] and the generalization to $d > 2$ was suggested by Johnson [8].

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

†The principal author to whom the correspondence should be addressed. E-mail address: `f.vitale@studenti.uninsubria.it`.

Traditionally, algorithm designers analyse the asymptotic number of primitive operations executed. For example, when referring to a heap data structure, the operations considered in a standard analysis are element comparisons and element moves. For many years this approach has dominated the development of algorithms and data structures. This approach has been shown to be insufficient and has led to evident deviations between the predicted performance and the experimental results, mainly because of reasons related to architectural details like pipelining, register spilling, and caching (see, for example, [1, 9]). Of these, caching effects are the most important.

Usually computer memory is composed of different random-access storage levels, abiding the principle of inclusion: the presence of a datum at level i implies the presence of the same datum at level $i+1$. Level i is assumed to be faster and smaller than level $i+1$. Each level is divided into disjoint and consecutive ***blocks*** of the same size, but block dimensions at different levels can vary. With the term ***cache*** we indicate the levels bridging main memory and the CPU. The cache stores the copy of a fraction of the data held in main memory that is likely to be requested by the CPU. When a data request cannot be satisfied by the cache, i.e. in the case of a ***miss***, a contiguous block of memory containing the data is copied from main memory to the cache, according to the replacement strategy adopted. We assume that when a miss occurs the block is copied from main memory to the cache following the ***least-recently-used*** (***LRU***) block-replacement policy. Furthermore we assume that the cache is ***fully-associative*** so that the block can be placed to any cache location. For further details on the architecture of computer memories, we refer to [7].

Since in most contemporary computers an access to main memory (and a block transfer from it) is more expensive than other primitive operations, caching effects cannot be ignored in the design of algorithms and data structures. A ***cache-aware*** algorithm needs to know the memory characteristics of the computer on which it is executed.A ***cache-oblivious*** algorithm, as defined in [5], is able to reach a good cache behavior without knowing architectural details of the cache.

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

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

formance with respect to the number of element comparisons and element moves performed. Moreover, the data structure still operates in-place without requiring any extra memory.

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

This definitions seem to state that cache oblivious algorithms are in general to be preferred to cache aware algorithms. However, in the attempt of get a good cache use, several cache oblivious algorithms and data structures involve the execution of so complicated sequences of additional instructions, e.g. sophisticated index manipulations, that the number of operations considered significant by the traditional increases in a considerable way, worsening the general performances. For example, in the applications of the layout for the cache oblivious search trees discussed in [10] is analysed the well known heap data structure in which the locations of the nodes are joined together with a global recursive technique (if n is the number of elements $O(\log \log n)$ levels of recursions). In spite of the good use of the cache and the complete independence from the architecture, the employment of loops with a non-constant number of iterations inside the function code for the calculation of node indices makes the data structure only of theoretical interest. Moreover, *pop*() in the worst case requires many element comparisons and element moves $(2 \log_2 n+O(1))$, whereas $\log_2 n+\log_2^* n+O(1)$ comparisons are sufficient for deleting a minimum element from a binary heap (Gonnet and Munro [6], corrected by Carlsson [2]).

The remainder of the report consists of five sections. In Section 2 we describe the data structure and we analyse the performances of the main operations, comparing them with some classical heap versions. In Section 3 we explain the implementation techniques that we employed in the programming phase. In Section 4 we describe an alternative realization of the priority queue able to reduce the overhead for navigate the data structure. In Section 5 we show the experimental results relative to our implementations and the conclusions are presented in Section 6 .

2. Design and analysis of the data structure

We will now briefly recall the principal variants of the algorithms for the execution of the main operations on the classical heap data structures. We will

also show the results known for the asymptotic analysis of the relative worst case performances for what concerns the number of element comparisons, element moves, and cache misses.

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

n: number of elements currently present in the heap.

M: cache capacity.

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

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

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

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

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

We also assume that the reader it is familiar with binary heap implementation and analysis. For specific details we refer to [3].

2.1 push(x)

2.1.1 Bottom-up execution

The path connecting the location $A[n]$ and the root is traversed in a bottom-up manner, copying every element to its child location until the first element equal or greater than x is found or until the root is reached. The new element is copied into the child location of this element.

Binary heap

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

Multiary heap

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

2.1.2 Binary search execution

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

Binary heap

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

Multiary heap

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

2.2 top()

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

2.3 pop()

2.3.1 Top-down execution

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

Binary heap

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

Multiary heap

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

2.3.2 Bottom-up execution

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

Binary heap

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

Multiary heap

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

Multiary heaps are able to get a better cache behavior compared with the binary heaps [9]. However the number of comparisons increases quickly with the value of d. This fact motivated us in the search of an improvement of the binary heap layout in order to limit the number of comparisons reducing at the same time the number of cache misses for the worst case. We called

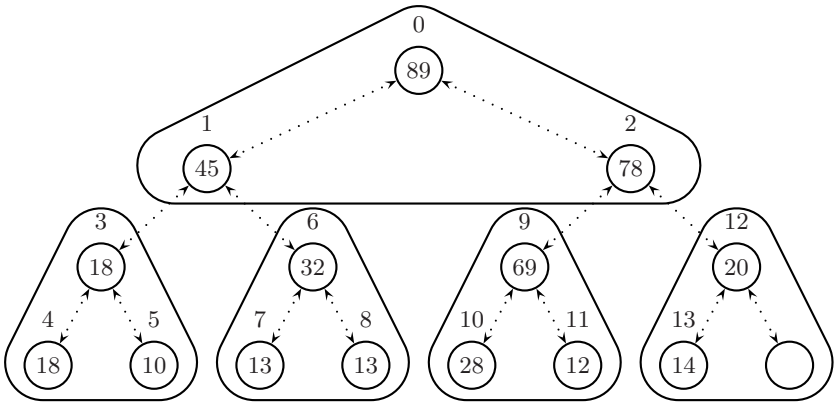

Figure 1. A 1-local heap of size 14.

this kind of heap h-local heap.

2.4 Local heap description

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1) we traverse top-down the in-node heap copying, to the location of each element on the path traversed, its greatest child (until we arrive to the bottom-level of the fat-node).
2) we compare the greatest child c of the last element copied with $A[n-1]$ (currently positioned as the last leaf in the last fat-node of the data structure);
   - if $A[n-1]$ is not the biggest we simply copy c to its parent position and we continue the traversing
   - otherwise, since we know now that the correct position of x is inside the last fat-node modified, we traverse it bottom-up copying every element smaller than $A[n-1]$ to its child position on the path traversed. We terminate inserting the last element $A[n-1]$.

2.5 Local heap analysis

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

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

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

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

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

push()

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

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

for the bottom-up execution.

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

pop()
The traversing of a fat-node requires $h+2$ comparisons. Since the number of fat-node on the path from any leaf until the root is $(\log_2 n)/(h+1)+O(1)$, $(1+1/(h+1))\log_2 n+O(h)$ element comparisons and $\log_2 n+O(h)$ element moves are executed. The worst case number of cache misses varies according to the parameters h and k. The worst case number of cache misses involved considering a single in-node heap is obtained if the last two elements belong to different blocks, and it is necessary to traverse it on its rightmost path. The root of such in-node heap contained in a fat-node on the rightmost path and the root of the in-node heap contained in its sibling fat-node belong to different blocks if and only if $h \geq \log_2 k-1$. It is now easy to see that the number of cache misses is:
$k = 1$: $2\log_2 n+O(h)$.
$k = 2$: $(2-1/(h+1))\log_2 n+O(h)$.
$h < \log_2 k-1$: $(2/(h+1))\log_2 n+O(1)$.
$h \geq \log_2 k$ and $k > 2$: $((2h-2\log_2 k\ +4)/(h+1))\log_2 n+O(h)$.
$h = \log_2 k-1$: $(3/(h+1))\log_2 n+O(1)$.
In the alternative execution employing the binary search along the path following the greatest child of each element for find the correct place of $A[n-1]$, the number of element comparisons becomes $\log_2 n+\log_2\log n+O(h)$ and number of element moves is $\log_2 n+h+O(1)$. We assume that M is at least $(2\log_2 n+2h+O(1))B$; the number of cache misses is again the same executed by the bottom-up *push*().

3. Implementation details

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

for *pop*() and sorting. In this section we analyze the code relative to the (top-down) *pop*() execution for $h = 3$ (shown in the appendices), because it is enough for explain in detail all the implementation techniques that we employed.

All the code is divided in three parts:

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

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

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

In Appendix Appendix A we show some of the policy function necessary in the *pop* implementation. We leave to the reader the comprehension of details (like template parameters) that are obvious or not important. The function *first_child*() simply returns the index of the first child of the element whose index is passed as parameter. It is easy to see that it uses the formulas relative to the $left\_child()$ function showed in the Section 2. In order to speed up the execution it stores the result of some arithmetical calculations in some variables that can easily be reused inside the rest of code. *top_all_present*() and *top_some_absent*() returns the index of the maximum child of the element whose index is passed as parameter (if all the children are present or if some children are missing).

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

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

We used a variable m containing the index of the root of the fat-node

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

Lines 021 and 022 calculates the index of the left children (root of a child fat-node) of the element whose index value is i. On 024 we have the if-else tree necessary for find correct location of element x (in the case in which it is inside the fat-node traversed). With this if-else tree we traverse up the in-node heap and moving back the elements modified previously.

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

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

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

The complete code of the local heap programs is accessible via the home page of the CPHSTL project [4].

4. An alternative realization

Analyzing the formulas used in the *parent*(), $left\_child()$, and *right_child*() functions showed in the Section 2, it easy to see that a remarkable part of the cost in term of time will concern the execution of divisions and multi-

plications of index values by F, the number of elements in a full fat-node.

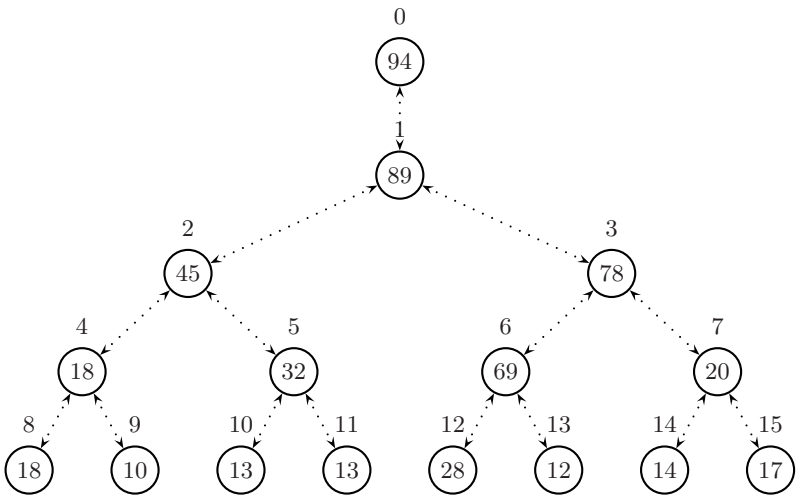

Figure 2. A complete extra-rooted heap.

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

In order to solve this problems and accelerate the execution of all the operations, it could be possible to consider an alternative kind of the fat-nodes. In this new approach, every fat-node will contain a modified heap that we will call *extra-rooted heap*. This kind of heap is simply composed by an ordinary binary heap in which the root will be (in breadth-first order) instead of the first, the second element of the whole structure, whereas the first location will be occupied by an element not smaller than any other element. We omit the formal definition about the relations among elements contained in different fat-nodes of this type, analogue to that one given in the Section 2.

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

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

approach from a practical point of view.

5. Experimental results

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

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

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

big integers : unsigned integers represented as strings of digits.

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

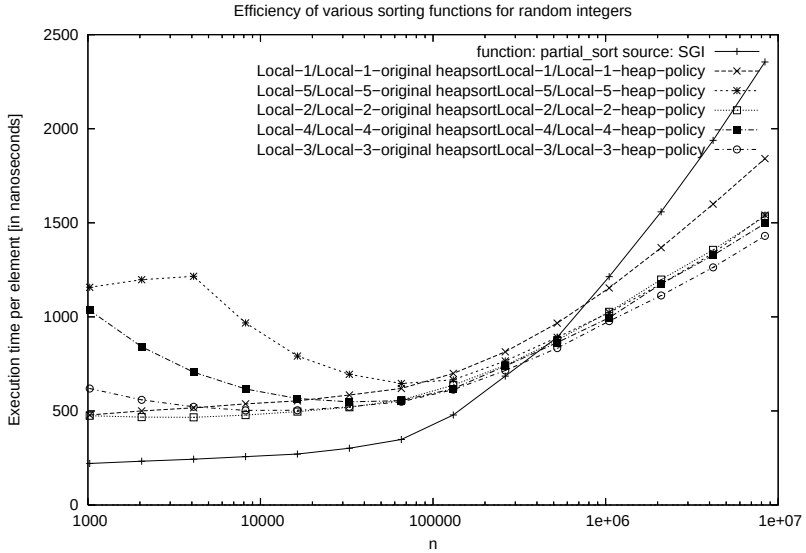

Figure 3. Sorting test for the h-local heaps and Silicon Graphics implementation (integers).

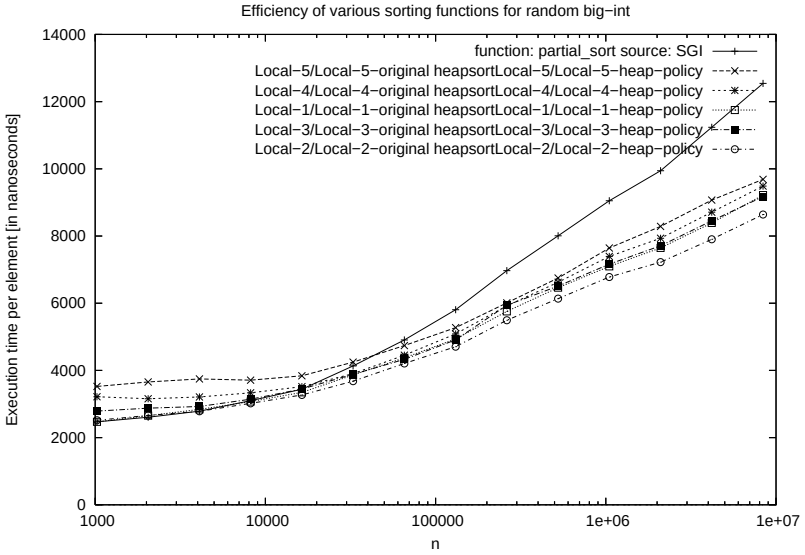

Figure 4. Sorting test for the h-local heaps and Silicon Graphics implementation (big integers).

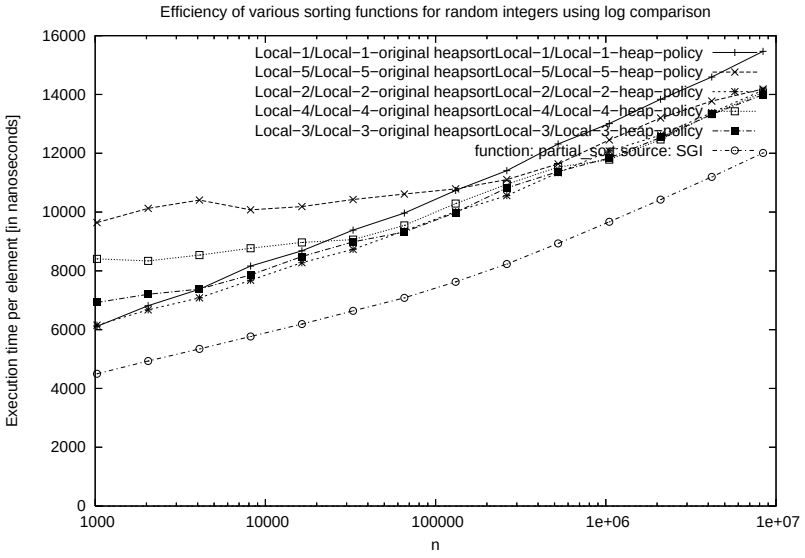

Figure 5. Sorting test for the h-local heaps and Silicon Graphics implementation (integers with logarithmic comparisons).

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

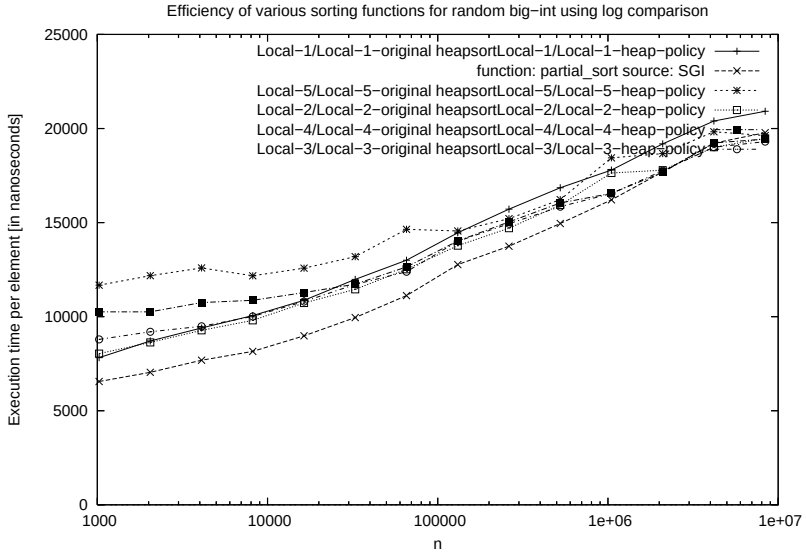

Figure 6. Sorting test for the h-local heaps and Silicon Graphics implementation (big integers with logarithmic comparisons).

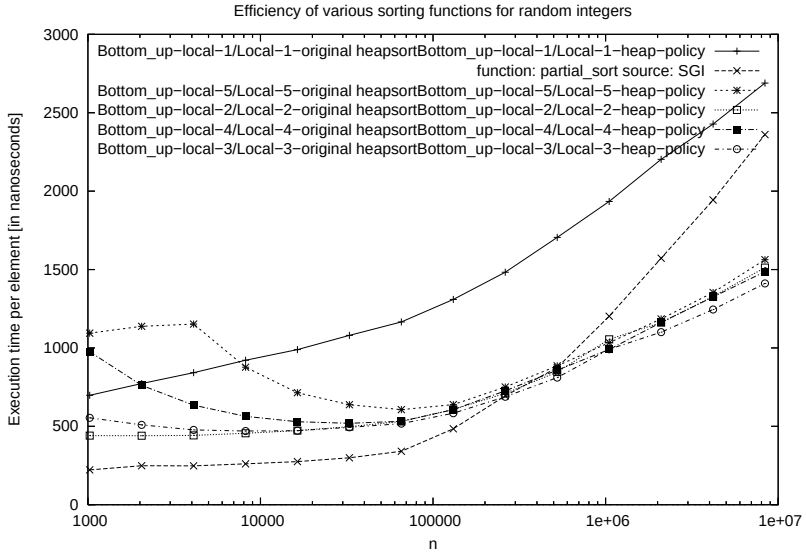

Figure 7. Sorting test for the h-local heaps and Silicon Graphics implementation with bottom-up approach (integers).

ference about the speed of the data structures.

As showed in the pictures, when we deal with a not too small number of elements, the performances for all the approach, except for integers with

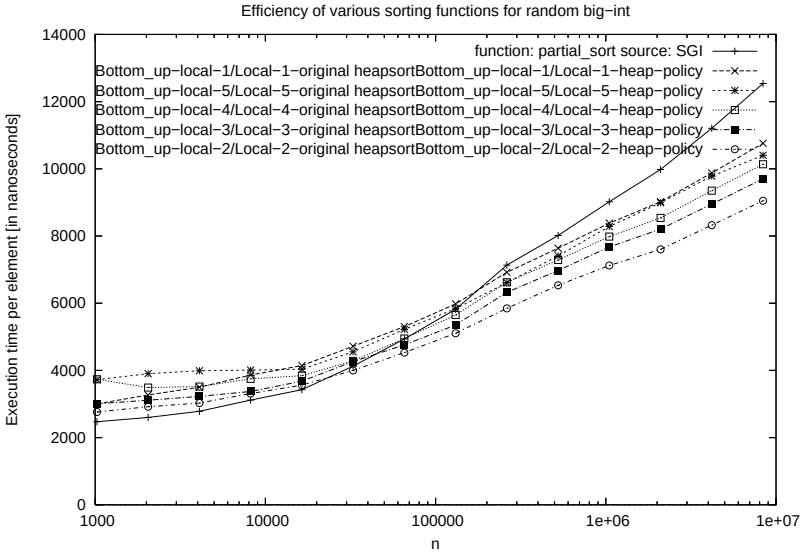

Figure 8. Sorting test for the h-local heaps and Silicon Graphics implementation with bottom-up approach (big integers).

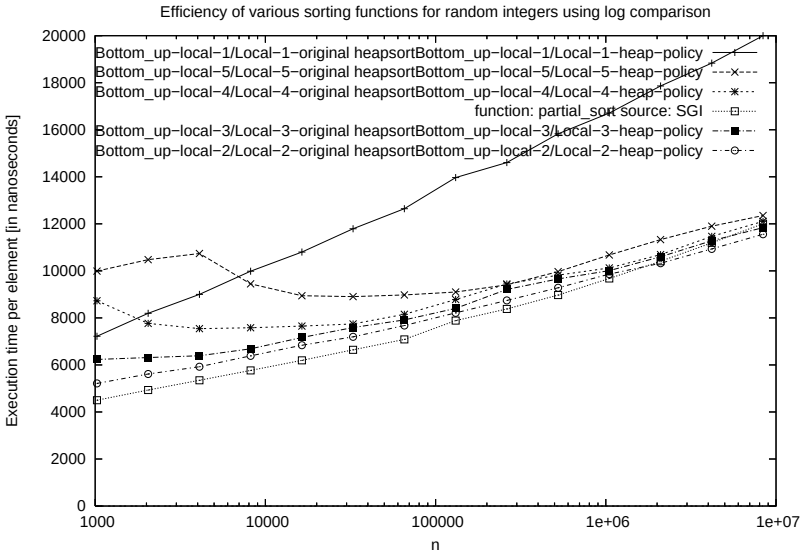

Figure 9. Sorting test for the h-local heaps and Silicon Graphics implementation with bottom-up approach (integers with logarithmic comparisons).

logarithmic comparison (and for the extra-rooted heap with logarithmic comparisons), result better than those relative to the classic binary heap, especially for $h = 3$ and $h = 2$.

The approach employing extra-rooted fat-nodes results slightly better for

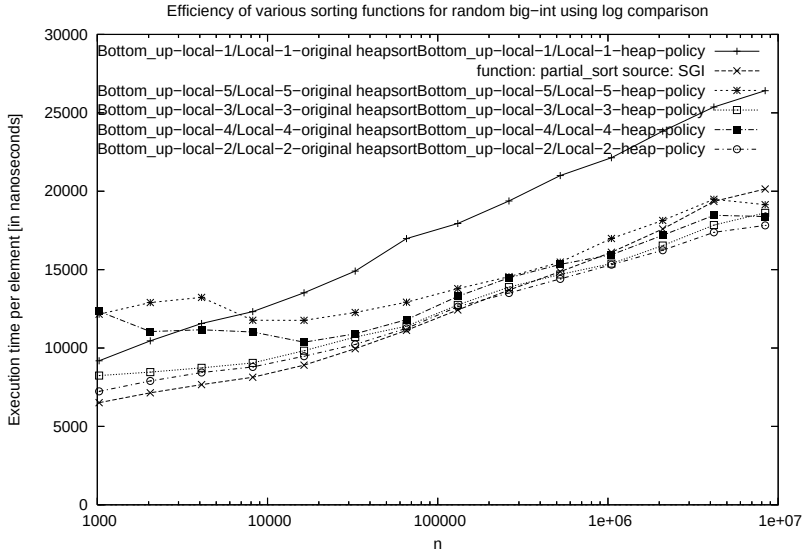

Figure 10. Sorting test for the h-local heaps and Silicon Graphics implementation with bottom-up approach (big integers with logarithmic comparisons).

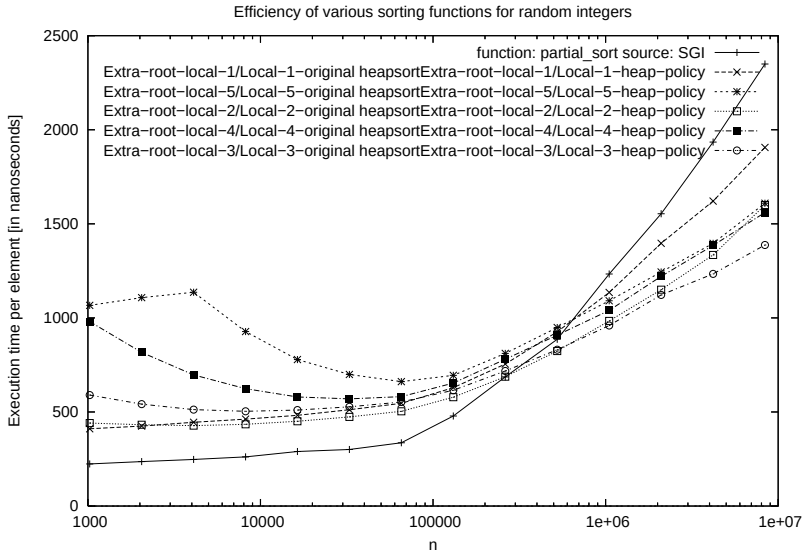

Figure 11. Sorting test for the extra-rooted h-local heaps and Silicon Graphics implementation (integers).

integers than the top-down normal approach, is spite of the growing of the height of the heap. For the dimension of the fat-nodes, the value $2^{h+1}-1 = 7$ or $2^{h+1}-1 = 15$ ($2^{h+1} = 8$ or $2^{h+1} = 16$ for the extra-rooted version) is therefore enough to get the best performances on the computer on which we

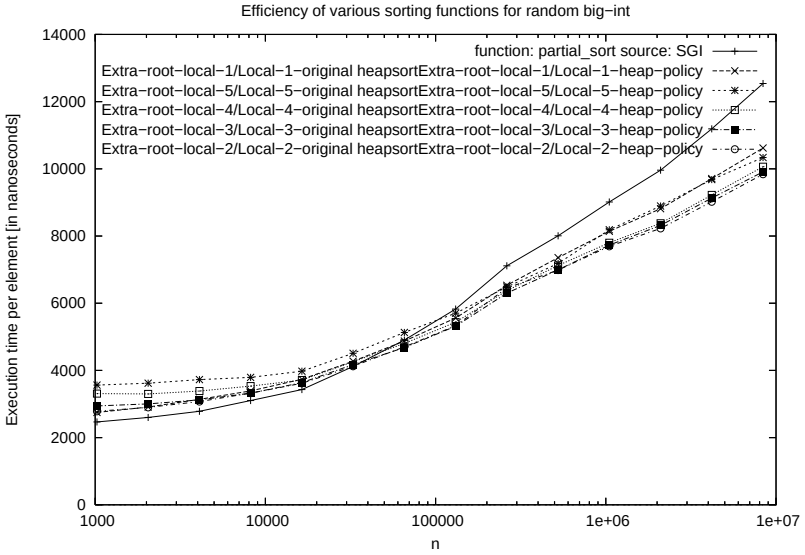

Figure 12. Sorting test for the extra-rooted h-local heaps and Silicon Graphics implementation (big integers).

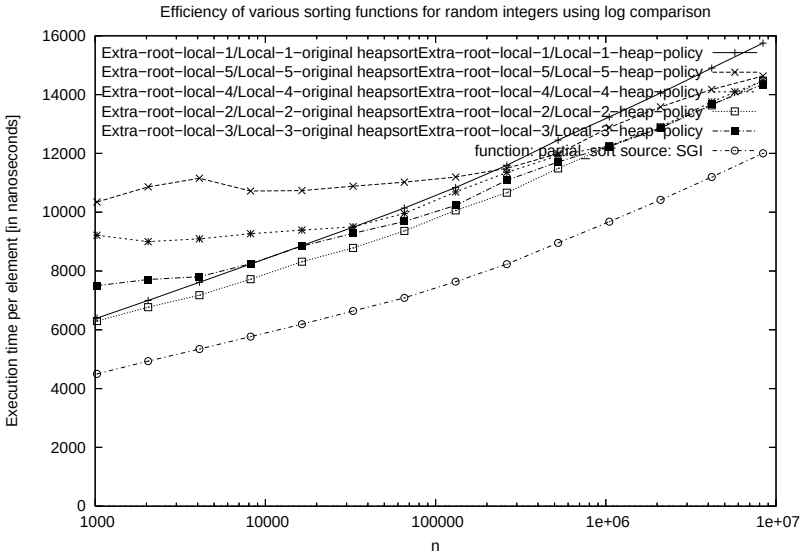

Figure 13. Sorting test for the extra-rooted h-local heaps and Silicon Graphics implementation (integers with logarithmic comparisons).

tested our implementations (AMD Athlon(tm) running with 1032.536 CPU MHZ and 256 KB of cache size).

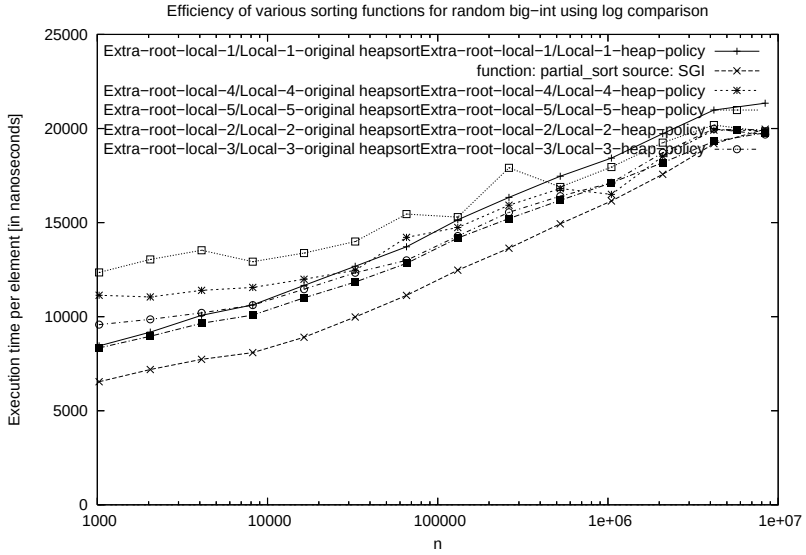

Figure 14. Sorting test for the extra-rooted h-local heaps and Silicon Graphics implementation (big integers with logarithmic comparisons).

6. Conclusions

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

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

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

Appendix A. Policy function code

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

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

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

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

Appendix B. sift_down() function code

```
001 template <typename position, typename index, typename element,
002    typename ordering, typename policy>
003 void
004 sift_down(position a, index i, element x, const ordering& less, policy& p)
{
005    index j;
006    index m = 15 * (p.last_leaf() / 240);
007    while (i < m) {
008       index k = i;
009       position ak = a + i;
010       i = (less(*(ak + 1), *(ak + 2))) ?  2 :  1;
011       *(ak) = *(ak + i);
012       j = i;
013       i = (i << 1) + 1;
014       if(less(*(ak + i), *(ak + i + 1))) ++i;
015       *(ak + j) = *(ak + i);
016       j = i;
017       i = (i << 1) + 1;
018       if(less(*(ak + i), *(ak + i + 1))) ++i;
019       *(ak + j) = *(ak + i);
020       j = i;
021       i += k;
022       i = (30 * i) - 195 - (210 * (k / 15));
023       if(less(*(a + i), *(a + i + 15))) i += 15;

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

```
038          else
039             *(ak + j) = x;
040          return;
041       }
042       else
043          *(ak + j) = *(a + i);
044    }

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

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

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

Appendix C. Heap function code

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

005    index i = 0;
006    index j = p.last_leaf();
007    element x = *(a + j);
008    *(a + j) = *(a + i);
009    p.erase_last_leaf(a);
010    sift_down<d, position, index, element, ordering, policy>(a, i, x, less,
p);
011 }
```

References

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

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

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

[4] Department of Computing, University of Copenhagen, The CPH STL, Website accessible at `http://www.cphstl.dk/` (2000–2005).

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

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

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

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

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

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

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