All-in-one implementation framework for binary heaps

Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark

Abstract. By a few search-engine queries, it was easy to identify several alternative ways of implementing a binary heap, the fundamental priority-queue structure loved by us all. Which one of these alternatives is the best in practice? The opinions of crowd-pullers and textbook authors are aligned: use an array. Of course, the correct answer is "it depends". To get from opinions to facts, an adaptable component framework was written that provides a variety of customization options so it could be used to realize many of the proposed variants. Also, some of the derived implementations were performance benchmarked. From this work, two conclusions can be drawn: (1) It is difficult to achieve space efficiency and speed at the same time. If n denotes the ***current*** number of values in the data structure, ε is a small positive real, $\varepsilon < 1$, and $|\mathcal{V}|$ denotes the size of the values of type $\mathcal{V}$ in bytes, space efficiency means $(1 + \varepsilon)|\mathcal{V}|n$ bytes of space, and speed means $O(\lg n)$ worst-case time per `push` (`insert`) and `pop` (`extract-min`). (2) Sometimes a linked structure and clever programming is a viable option. If a binary-heap variant that you would need is not available at the software library you are using, by reading this essay you can be spared from some headaches.

1. Introduction

In its elementary form, a priority queue `Q` is a data structure that stores a multiset of values and supports the operations:

top: Return a reference to the value in `Q` that has the highest priority with respect to some predetermined comparison function. Precondition: `Q` is not empty.

push: Insert a new value into `Q`.

pop: Remove the value having the highest priority from `Q`. Precondition: `Q` is not empty.

Naturally, there should be a way to construct `Q`, given a sequence of values, and to destroy `Q` when it is not needed any more. To be of any use, there should also be function `size` that returns the number of values in `Q`. Throughout the text, n denotes the number of values stored in a data structure prior to the operation in question and $\lg n$ is a shorthand for $\log_2(\max\{2, n\})$.

It is widely agreed that the simplest and most practical way of implementing a priority queue is to use a binary heap [36], which is described in

most textbooks on data structures and algorithms (see, e.g. [10, Section 6]). Recall that a ***binary heap*** has the following properties:

- It is a ***nearly-complete binary tree*** [10, Section B.5.3], i.e. a tree that is obtained from a complete binary tree by removing some of the rightmost nodes at the bottommost level of the tree.
- Each node of this tree stores a value.
- The values are ordered such that, for each node, the value stored at that node has a higher priority than the values in the descendants of that node (if any).

Normally, the tree is represented in an array, but other representations are also possible.

In this essay, different options of implementing a binary heap are examined. For a recent survey on priority queues in general and their theoretical characteristics, see [6]. The following question-answer (***Q-A***) pair sums up the contents of this essay.

Q: What is the best way of implementing a binary heap in a software library?

Here the word *best* is intentionally ambiguous.

A: Provide a generic framework that can be used to realize a wide variety of implementations, and let the user of the library select the implementation that suits best for her or his needs.

The focus of this essay is on the design, implementation, and benchmarking of such all-in-one framework for realizing different binary-heap variants, whether they were array-based, pointer-based, or a combination of the two. An important goal is to make the framework customizable and extendable. For other adaptable component frameworks of its kind, see [2, 29] (search trees), [21] (dynamic arrays), and [8, 12] (addressable priority queues).

I decided to investigate this matter because I doubted whether the binary-heap framework used in our earlier experimental studies [8, 12] was good enough. In general, binary heaps performed well in these horse races, but both of these earlier studies failed to broach two issues:

(1) In policy-based benchmarking, frameworks are used to achieve fair results. Sometimes a framework can be too general, sometimes too fine-grained. In the case of our experimental papers, it remained unresolved how big the overhead caused by the frameworks was.

(2) Our binary-heap implementation failed to support `push` in $O(\lg n)$ worst-case time and guarantee that the amount of space used was $O(n)$ at all times. This was because the dynamic-array implementation used in the experiments did not guarantee good worst-case runtime or space efficiency.

In most textbooks on data structures and algorithms an array-based solution is described. However, if you study the proposed implementations carefully, you will observe that ***not many*** of them support `push` and `pop` in $O(\lg n)$ worst-case time, or use $O(n)$ space, because the dynamization of the

underlying array is not done properly. At this point, you should be worried and ask why this has not been done correctly.

An exception is the book by Goodrich et al. [16, Section 7.3.3] where the authors sketch a generic implementation that can realize both an array-based and a pointer-based binary heap. However, the implementation details of the pointer-based solution were left to the exercises. The central problem is how to maintain a pointer to the last node at the bottommost level of the heap. They offered three options:

threading [16, Exercise C-7.7]***:*** Store additional pointers at the nodes such that each node having no children has direct access to its predecessor and successor in breadth-first order. These pointers can be maintained such that the extra overhead is just $O(1)$ per `push` and `pop`.

bit stack [16, Exercise C-7.8]***:*** Use the bits in the binary representation of $n+1$ (`push`) or n (`pop`) to access the last node when traversing the tree downward starting from the root, 0 meaning go to the left child and 1 meaning go to the right child. The bit string is processed from the most significant to the least significant and the first 1 refers to the root.

finger search [16, Exercise C-7.9]***:*** Search for the predecessor or (at this point non-existing) successor of the last node by traversing the tree first bottom up and then top down starting from the present last node.

Which of these alternatives will work best or is there even a better way of implementing a binary heap as a linked structure?

Based on some initial googling, the question how to implement a binary heap seems to be of interest for many bloggers, programmers, and students (and indirectly their teachers). With the keywords "pointer-based binary heaps", Google ranked highest a page [31] at Stack Overflow. People, who wrote on this page, mentioned several implementation alternatives; my interpretation of the options that were considered feasible was as follows:

implicit tree: a standard implementation using an array (3 votes)

referent tree: an array of pointers to nodes storing the values (3 votes)

linked tree: a pointer-based tree implementation à la binary search trees (9 votes); some specific variants were supported explicitly: threading (1 vote), bit stack (1 vote), and finger search (3 votes).

In the answer to the same Google query, on the page [30] ranked third, there was a beautiful sketch how to implement a pointer-based solution using a bit stack and storing two pointers per node. However, no one could convincingly justify why and when one implementation would be better than another.

The structure of this essay follows Kolb's learning cycle [23], according to which in every learning situation one should answer the following four questions: why (Section 3), what (Section 4), how (Section 5), and what if (Section 7). I added two more questions to this list: to whom (Section 2) and how well (Section 6). After answering these questions, I conclude the essay with a couple of remarks (Section 8).

```
#include <functional> // std::less
#include <vector> // std::vector

template <typename 𝒱, typename 𝒮 = std::vector<𝒱>, typename 𝒞 = std::less<𝒱>>
class priority_queue {
public:

  using value_type = 𝒱;
  using container_type = 𝒮;
  using ℕ = typename 𝒮::size_type;

  priority_queue(𝒞 const&, 𝒮 const&);
  ~priority_queue();

  ℕ size() const;
  𝒱 const& top() const;
  void push(𝒱 const&);
  void pop();
};
```

Figure 1. Part of the interface of the C++ standard-library priority-queue class template

2. To whom?

I expect that the reader of this essay has taken a course on programming and another on algorithmics either at university or high-school level. Possibly, this essay could also be used as a supplementary material on an algorithmics course. All programs are described using C++ [33] so the reader should be able to read them. The text may also inspire teachers when lecturing on priority queues, textbook authors when writing a chapter on priority queues, researchers when seeking directions for future research, and professional programmers when designing frameworks.

3. Why?

In the C++ standard library [9, Clause 23.6.4], a priority queue is a class template that has three type parameters (see Figure 1):

𝒱: the type of the values stored,

𝒮: the type of the sequence used for storing the values, and

𝒞: the type of the comparator used in value comparisons.

In Figure 1, as in the other transliterated programs, special symbols are used to distinguish type names from variable names. When instantiating `std::priority_queue`, the argument corresponding to $\mathcal{V}$ can be any type supporting copy construction and copy assignment (or move construction and move assignment). As to $\mathcal{S}$, the argument can be any sequence supporting copy construction, **operator**[], `size`, `push_back`, and `pop_back`. And as to $\mathcal{C}$, the semantic requirement is that the function or functor given as argument induces a strict weak ordering on the values [9, Clause 25.4]. By default,

the sequence is of type `std::vector`, which is a dynamic array implemented in any incarnation of the C++ standard library, and the comparator is of type `std::less`, which makes the underlying priority queue a max-heap.

One of the cornerstones in the design of the C++ standard library was copy semantics [32]. That is, values are owned by a container; they are never shared by different containers. Later, the standard library was extended to support move semantics as well [9]. This is in harmony with the original design since a value is still owned by a single container, but the owner can change. If you are unsure about the details of move semantics, consult any up-to-date textbook on C++ for more information (e.g. [33, Section 17.5]).

Often `std::priority_queue` is implemented as a binary heap and it is well programmed, so the default set-up is satisfactory in many applications. Let us list some situations where this set-up is no longer sufficient.

tight worst-case guarantees: The underlying sequence, as it is often the case with the implementations of `std::vector`, may not support `push_back` and `pop_back` in $O(1)$ worst-case time. As a consequence, the time bounds for `push` and `pop` may just be $O(\lg n)$ in the amortized sense.

space efficiency: When memory allocation and freeing is done piecewise, in addition to the space required by the n values, $\Theta(\sqrt{n})$ extra space is known to be necessary and sufficient for a binary heap [7]. In the default set-up, the amount of extra space used can be much higher.

expensive moves: $\sim\lg n$ value moves per `push` or `pop` can be too much. Here $\sim f(n)$ means the quantity that approaches $f(n)$ when n grows to infinity.

expensive comparisons: $\sim\lg n$ value comparisons per `push` or $\sim 2\lg n$ value comparisons per `pop` can be too much. On the other hand, in the average case the default set-up performs almost optimally in this respect.

shared values: Sometimes a value is a member of several sets at the same time. It can be tedious to link the copies having different owners, and it can be inefficient to traverse back and forth between the copies.

larger operation repertoire: If a user needs support for a general `extract`, taking the position of a value within the data structure as its parameter, the default set-up relying on an array of values cannot be used at all. Yet another interesting operation is `merge`, which combines two priority queues, and arrays are not well suited for that.

referential integrity: Because a straightforward implementation moves values around, references to them cannot be kept valid, and handles to values within the data structure cannot be provided (compare the previous two items).

exception safety: The template arguments are specified by the user and their operations can throw exceptions. If an exception occurs in the middle of a modifying operation, the data structure may be left in an inconsistent state.

concurrent updates: With off-the-shelf algorithms, deadlocks can occur if several processes update a binary heap concurrently.

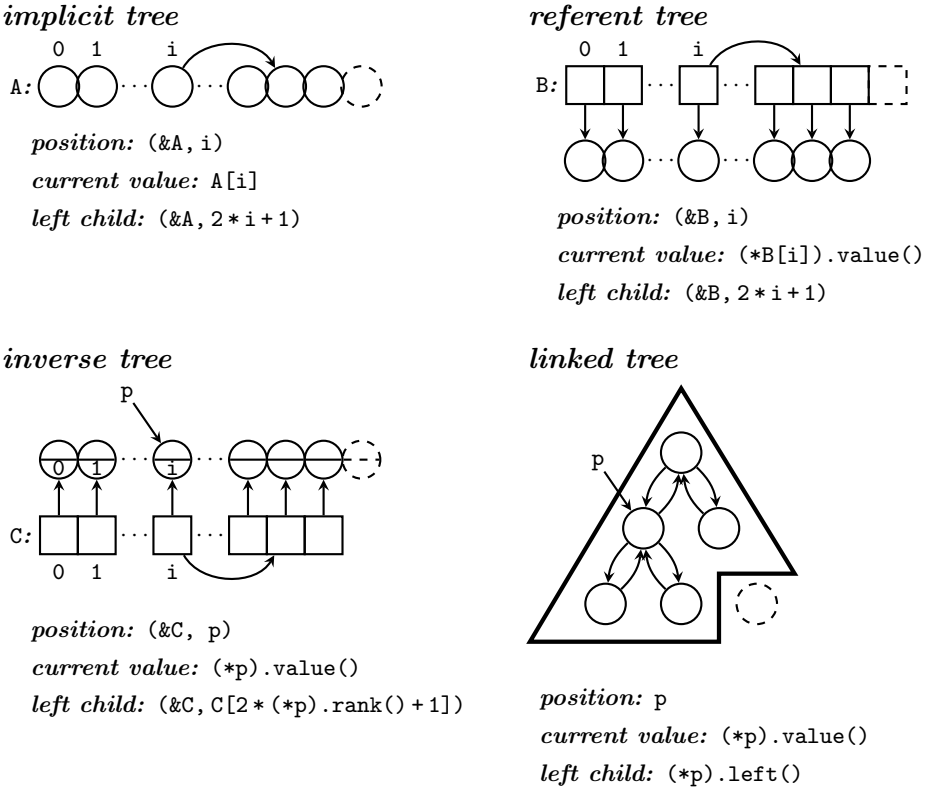

Figure 2. Visualization of the four desirable implementations; values live in circles

A priority queue is a fundamental data structure that is needed in many applications and, as the above list confirms, the demands in different applications vary. Since there are situations where the standard-library implementation does not fit the bill, it is necessary to consider other options.

The implementations of interest for this essay are visualized in Figure 2. In the approaches that use an array, the array can store the values directly (***implicit tree***), it can store pointers to cells that each encapsulate a value (***referent tree***), or an array can be used for navigational purposes so that the values are stored in cells, each cell stores a reference to an array entry, and that entry stores a pointer back to the corresponding cell (***inverse tree***) [10, Section 6.5]. The fourth approach is to use a purely linked structure (***linked tree***). In [8, 12], the inverse-tree approach was benchmarked against its competitors; the given solution also supported `extract` in logarithm worst-case time. In [16], both the implicit-tree and linked-tree approaches were

implicit tree

```
A: array of values
   capacity fixed
   indexing starts from 1
n: size of A
in: value to be added
i, j: indices
less: comparator

void inheap(A, n, in) {
  i = n = n + 1;
scan:
  if (i > 1) {
    j = i / 2;
    if (less(A[j], in)) {
      A[i] = A[j];
      i = j;
      goto scan;
    }
  }
  A[i] = in;
}
```

referent tree

```
first, past: random–access iterators
less: comparator
in: value to be added
hole, parent: indices

void push_heap(first, past, less) {
  in = *(past − 1);
  hole = past − first − 1;
  parent = (hole − 1) / 2;
  while (hole > 0 and less(*(first + parent), in)) {
    *(first + hole) = *(first + parent);
    hole = parent;
    parent = (hole − 1) / 2;
  }
  *(first + hole) = in;
}

in: value to be added
p: pointer to a cell encapsulating a value
D: dynamic array of pointers to cells
   indexing starts from 0
less_ref: comparator comparing pointers to
   cells using values

void push(in) {
  p = create(in);
  D.push_back(p);
  push_heap(D.begin(), D.end(), less_ref);
}
```

Figure 3. Stylized implementations of push; these programs are freely adapted from the sources given; for the functions not described, consult the sources; (1) implicit tree [36], (2) referent tree (programmed by Jensen [11]), (3) inverse tree [18], and (4) linked tree [16]

inverse tree

```
D: dynamic array of pointers to
   cells referring back to D
i, j: indices
less: comparator

void siftup(D, j, less) {
  i = j / 2;
  while (j ≠ 1 and
    less(D[i]→value(), D[j]→value())) {
    swap_cells(D[i], D[j]);
    j = i;
    i = j / 2;
  }
}

pair: (pointer to the owner, pointer to
   a cell) specifying a position
D: dynamic array of pointers to cells
   indexing starts from 1
p: pointer to a cell storing a value and
   an index back to D
last: index
H: heapifier that does all data–structural
   transformations

position insert(pair) {
  get<0>(pair) = &D;
  p = get<1>(pair);
  D.push_back(p);
  last = D.size();
  (*p).set_rank(last, &D);
  H.siftup(D, last, less);
  return pair;
}

in: value to be added
p: pointer to a cell storing a value and
   a reference back to its owner
t: iterator encapsulating a position
R: realizator operating with cells instead
   of values

iterator push(in) {
  p = create(in);
  t(p);
  t = R.insert(t); // casts automatic
  return t;
}
```

linked tree

```
in: value to be added
u, z: positions of cells storing values
T: tree of cells
less: comparator

void insert(in) {
  z = T.add(in);
  while (not T.is_root(z)) {
    u = T.parent(z);
    if (not less(z.value(), u.value())) {
      break;
    }
    T.swap_values(u, z);
    z = u;
  }
}
```

Figure 3. *(cont.)*

realized simultaneously by the same class template, but this solution did not support `extract` since values were moved. In order to keep all cell references valid, it is better (but more expensive) to swap cells, instead of values. In our implementations the cell references are retained valid if possible.

Actually, the desired implementations are available at the CPH STL [11]. Figure 3 shows, in a stylized form, how `push` is programmed in these implementations. Can you see that the insertion algorithm used in these four programs is the same? The goal of this study is to provide an implementation framework that can be used to generate different binary-heap variants. Hopefully, it would be easier to maintain this framework than four or more separate implementations. After reading the essay and possibly looking at the source code, you should evaluate if this goal is met.

The contribution of this essay can be summarized as follows:

- I provide an all-in-one framework that can be used to generate a large set of implementations for a binary heap.
- I explain how this framework can be customized—potentially extended—to derive the desired implementations.
- I report how well some of the basic implementations perform in my environment which provides guidelines for their use.
- I place the developed code in the public domain (see, "Software availability" at the end of the essay) so that other programmers do not need to reinvent the wheel.

4. What?

Let us try to understand what a binary heap is in its abstract form. After defining the terminology, the work done can be described concisely. Recall that the goal is to write a generic framework that can be easily customized and extended to get a wide variety of implementations for a binary heap.

Our way of thinking about data structures is depicted in Figure 4. A data structure is comprised of a set of ***cells***; each cell (illustrated as a circle in the figure) is capable of storing a ***value***. Every cell has a ***position***. A cell can be inside or outside a data structure, but—in both cases—it is still owned by that data structure. In addition to a value, a cell can store ***cell references*** (i.e. indices, references, pointers, or handles) to other cells. There are three kinds of special cells: (1) ***hole*** is an empty cell inside the data structure that does not store any value; (2) ***none*** is a fictive cell outside the data structure that does not exists at all; and (3) ***access point*** is a dummy cell outside the data structure which provides access to some specific cell inside the data structure. For all data structures of the same type, there is only one `none`. Therefore, for example, when two such data structures are merged, all cell references to `none` still refer to the same fictive cell after the merge.

Concretely, when values are stored directly in array `A`, each array entry is an invisible cell around the value. The position of such a cell is specified by a pair (`&A`, `i`), where `&A` is the start address of array `A` in memory and

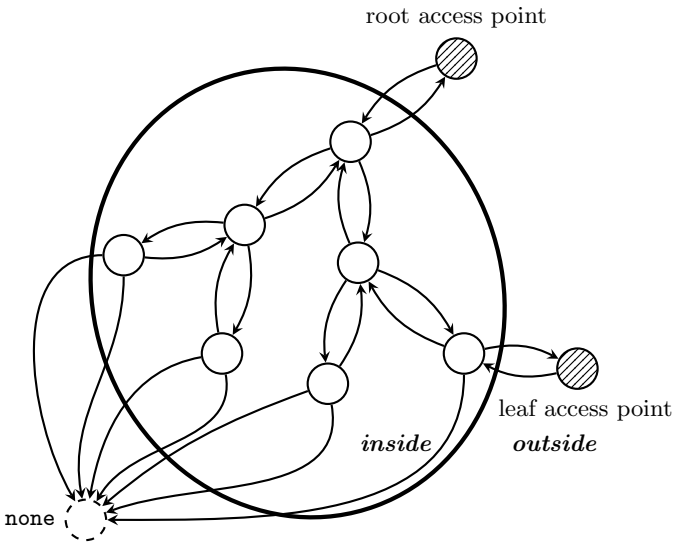

Figure 4. Navigational view of a data structure

`i` is the ***rank*** of a value telling how many values are stored before it in `A`. In a linked structure, the cell is a node and the position can be specified by giving the address of that node in memory.

This view is similar to that used in the C++ standard library, where the containers are sequences of values and values have positions. There an object encapsulating a position is called an iterator. We call this kind of object a ***navigator*** since there can be other operations than **operator**++, **operator**−−, **operator**+, or **operator**− that can give access to a neighbour of a cell. Unlike the standard, in our model, one cannot get back from `none` to the data structure; one has to use one of the access points. In spite of these differences, the important thing is that both an iterator and a navigator can be understood as a ***name*** of the cell they refer to. This leads to natural programs since, for example, a variable `root` can be used to name the root of a tree. In particular, `root` is not a pointer to, a reference to, or the rank of that cell; it is its name.

Although the interface of the binary-heap class template is identical to that of `std::priority_queue`, the concept requirements for the second type parameter are quite different. Let `T` be an object of that type. This data structure is a nearly-complete binary tree that maintains its cells in breadth-first order. So, in a broad sense, it is still a sequence. Observe that the operations to be supported on `T` do not directly operate with the data stored at the cells.

`size:` Return the number of cells in `T`.

`root:` Return the name of the first cell in `T`. Precondition: `T` is not empty.

`last:` Return the name of the last cell in `T`. Precondition: `T` is not empty.

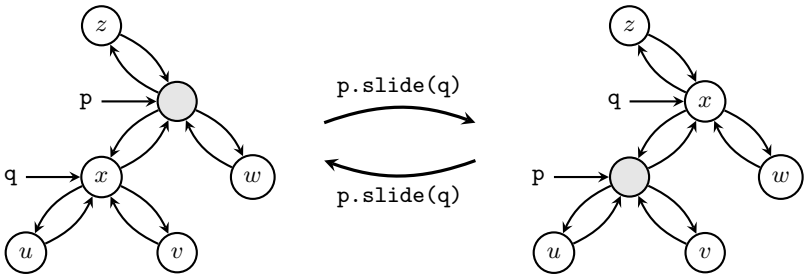

Figure 5. Illustrating `slide`; `p` refers to a hole and `q` to its neighbour

expand: Make `T` greater by adding a new hole immediately after the last cell of `T` and return the name of that cell. The cell references to and from the added cell are to be updated accordingly.

contract: Make `T` smaller by removing the last cell of `T`. The cell references to the removed cell must be updated accordingly. Precondition: `T` is not empty.

Any container (e.g. of type `std::vector`) that supports the operations `begin`, `end`, `push_back`, and `pop_back` could be adapted to realize `T`. The key issue is how to update the cell references in the modifying operations.

The main difference between `T` and a standard-library container is that `T` must be associated with a special navigator, not an iterator. In a sense, a navigator specifies shortcuts from a cell to some other cells. However, there is not necessarily a direct linkage between the cells, but a shortcut can be taken based on some computation. Let `p` and `q` be two navigators that both encapsulate a position of some cell. The operations to be supported include:

operator=: The assignment `p.`**operator**`=(q)` or `p = q` makes `p` refer to the same cell as `q`.

operator:* The dereference `p.`**operator**`*()` or `*p` returns a reference to the value stored at the cell referred to by `p`. Precondition: The cell is inside `T`.

operator== | *operator≠:* The comparison `p.`**operator**`==(q)` or `p == q` returns **true** if `p` and `q` refer to the same cell; otherwise, it returns **false**. The output of the comparison `p.`**operator**`≠(q)` is **not** `p.`**operator**`==(q)`.

left | *right* | *parent:* The operation `p.left()` returns the name of the left child of the cell referred to by `p`, or `none` if this leads outside `T`. Precondition: The cell is inside `T`. Correspondingly, the operations `p.right()` and `p.parent()` return the name of the right child and the parent.

slide: As the result of `p.slide(q)`, the positions of `p` and `q` are swapped in `T`. The cell references in these two cells and their neighbours are to be updated accordingly. Preconditions: The cell referred to by `p` is a hole and the cell referred to by `q` is a neighbour of that hole. This structural transformation is illustrated in Figure 5.

swap: As the result of `p.swap(q)`, the positions of `p` and `q` are swapped in `T`. The cell references in and around the swapped cells are to be updated

Table 1. Theoretical properties of the binary-heap variants considered. The space bounds are in bytes. Here $|\mathcal{V}|$, $|\mathbb{N}|$, and $|\mathcal{N}*|$ denote the size of the values of type $\mathcal{V}$, integers of type $\mathbb{N}$, and pointers to nodes of type $\mathcal{N}$ in bytes, respectively. In our test set-up, $|\mathcal{V}| = 4$, $|\mathbb{N}| = 8$, $|\mathcal{N}*| = 8$, and $\varepsilon = \frac{3}{256}$. The alignment cost (given in brackets) induced by `std::allocator` was measured experimentally using the space-cost micro-benchmark from Bentley's book [3, Appendix 3]. All variants support `top` in $O(1)$ worst-case time, and `push` and `pop` in $O(\lg n)$ worst-case time, n being the size of the data structure prior to each operation. The worst-case complexity of `extract` is given below. The minus sign − means that this operation cannot be supported efficiently.

variant	***space*** / n ***as*** $n \to \infty$	***value moves per*** `push`/`pop`	`extract`
implicit tree			
resizable array	$6\|\mathcal{V}\|$	$\sim\lg n$	−
pile	$2\|\mathcal{V}\|$	$\sim\lg n$	−
sliced array	$\|\mathcal{V}\| + \varepsilon\|\mathcal{V}*\|$	$\sim\lg n$	−
referent tree			
resizable array	$6\|\mathcal{N}*\| + \|\mathcal{V}\| + \Delta$ $[\Delta = 28]$	1 [a]	−
pile	$2\|\mathcal{N}*\| + \|\mathcal{V}\| + \Delta$ $[\Delta = 28]$	1 [a]	−
sliced array	$\|\mathcal{N}*\| + \varepsilon\|\mathcal{N}**\| + \|\mathcal{V}\| + \Delta$ $[\Delta = 28]$	1 [a]	−
inverse tree			
resizable array	$6\|\mathcal{N}*\| + \|\mathcal{V}\| + \|\mathbb{N}\| + \Delta$ $[\Delta = 20]$	1 [a]	$O(\lg n)$
pile	$2\|\mathcal{N}*\| + \|\mathcal{V}\| + \|\mathbb{N}\| + \Delta$ $[\Delta = 20]$	1 [a]	$O(\lg n)$
sliced array	$\|\mathcal{N}*\| + \varepsilon\|\mathcal{N}**\| + \|\mathcal{V}\| + \|\mathbb{N}\| + \Delta$ $[\Delta = 20]$	1 [a]	$O(\lg n)$
linked tree			
threading	$4\|\mathcal{N}*\| + \|\mathcal{V}\| + \Delta$ $[\Delta = 12]$	1	$O(\lg n)$
bit stack [30]	$2\|\mathcal{N}*\| + \|\mathcal{V}\| + \Delta$ $[\Delta = 12]$	1	−
finger search	$3\|\mathcal{N}*\| + \|\mathcal{V}\| + \Delta$ $[\Delta = 20]$ [b]	1 [a]	$O(\lg n)$

[a] Our implementation performs one default construction and two move assignments per `push`, and two move assignments and one destruction per `pop`

[b] Our space-optimized implementation requires $\sim(2|\mathcal{N}*| + |\mathcal{V}| + \Delta)n$ bytes $[\Delta = 12]$

accordingly. Preconditions: The cell referred to by `p` is a hole and the two cells are not the same. This structural transformation is otherwise as `slide`, but the two cells need not be neighbours of each other.

The operations `left`, `right`, and `parent` have no side-effects as, for example, **operator**++ for iterators, whereas `slide` and `swap` do modify the inter-cell linkage.

The performance of the array-based approaches depends heavily on how the dynamization of the underlying array is done. Here I rely on our earlier work [19, 20, 21] and purposely only use implementations that support **operator**[], `push_back`, and `pop_back` in $O(1)$ worst-case time. The implementations used are taken from the CPH STL [11]:

resizable array: This solution is part of computing folklore so it is used as a baseline for other worst-case-efficient implementations. The values are split over at most two contiguous memory segments and the total size of these segments is adjusted to the number of values using doubling, halving, and incremental copying. A full description and analysis of this structure can be found, for example, from [13, Appendix A.1].

pile: More specifically, this refers to the levelwise-allocated pile described in [20]. The values are split over a logarithmic number of contiguous memory segments, which increase exponentially in size and of which only the last may be partially full. The realization of **operator**[] assumes that the whole-number logarithm of a positive integer can be computed in $O(1)$ worst-case time.

sliced array: This solution maintains a resizable array of pointers to contiguous memory segments, each of the same size, which gives its name. As above, only the last segment may be partially full.

With these worst-case-efficient dynamic arrays, the worst-case bounds proved for binary heaps in textbooks and other sources actually hold. The theoretical properties of the studied implementations are summarized in Table 1 so that the reader can grasp the big picture quickly. Based on the theoretical properties, I find it most interesting to know what is the relative performance of the inverse-tree and linked-tree approaches and how much slower they are compared to the implicit-tree approach.

5. How?

I developed the framework in a few sprints; the outcome of each was a package corresponding to one of the desirable implementations. Let us examine these packages one at a time.

5.1 Implicit structure

The UML structure diagram given in Figure 6 shows the involved classes including their type parameters, type members, friends, data members, and function members. The interconnections between the classes are specified using templates, no inheritance is used. Especially, the type aliases and **friend** declarations are helpful in order to understand the interconnections.

As to the sequence storing the values, any class template that has the same interface as `std::vector` can be used. This sequence combined with a rank navigator is used in the realization of the implicit-tree class template, which is then used by the binary-heap class template. As shown in the class diagram, only a small subset of the operations of `std::vector` is needed, but the implicit tree could be made more feature-rich if necessary.

The fact that the implicit-tree class template is an adaptor of the sequence class template is apparent from the implementation of the operations shown in Figure 7. Albeit innocent, in `expand` the value type should allow default construction, whereas, for example, `std::vector` does not make this requirement for its value type. Technically, this means that the value stored at a hole is valid but unspecified. In terms of efficiency, a hole could even be left uninitialized, but this is not in accordance with good programming practice. Note that our implicit-tree implementation supports a bit larger set of operations (copy/move construction, copy/move assignment, `swap`, and `clear`), but I do not discuss this boilerplate code here.

Figure 6. Classes involved in the implicit-tree package; + means that a member is public and − that it is private

```
𝒱& operator[](ℕ i) {
  return sequence[i];
}

ℕ size() const {
  return sequence.size();
}

ℛ root() {
  return ℛ(this, 0);
}

ℛ last() {
  return ℛ(this, size() − 1);
}

ℛ expand() {
  sequence.push_back(std::move(V()));
  return last();
}

void contract() {
  sequence.pop_back();
}
```

Figure 7. Kernel of the implicit-tree class template; $\mathcal{V}$, $\mathbb{N}$, and $\mathcal{R}$ are type aliases for the value type, size type, and rank-navigator type; `sequence` is the container of values in use

In a rank navigator, a position is represented by storing a (pointer, rank) pair; the pointer refers to the container—the ***owner***—that contains the value referred to and the rank is its index within the container. A navigator class template can be implemented in a similar manner as an iterator class template; the set of operations being supported is just different. The implementation of some of the operations is shown in Figure 8. Note that both `slide` and `swap` break the referential integrity by moving a value. We use `std ::move` to signal that the value can be moved by a move assignment, if such an operator is available. This will also indicate that the hole is moved. In the forthcoming packages, when the parameter of these functions is specified to be **const**, they are known to guarantee referential integrity.

When operating with navigators, compared to array indices, there is some overhead. First, in **operator**=, in addition to the rank, the pointer to the owner has to be assigned as well. Second, in `left`, `right`, and `parent`, an extra **if** statement is needed for determining whether a position gets outside the data structure or not. Third, in `slide` and `swap`, the ranks are swapped, but the other navigator will be disregarded by the calling routine after performing these operations. Fourth, in **operator**==, in addition to the ranks, it should also be checked whether the owners are the same. On the other hand, since the default constructor can be used to create `none`, it is easy to determine whether or not a navigator refers to `none` using **operator**==.

```
V̌& operator∗() const {
  return (∗owner_pointer)[rank];
}

R left() const {
  N child = 2 ∗ rank + 1;
  if (child ⩾ (∗owner_pointer).size()) {
    return R();
  }
  return R(owner_pointer, child);
}

R right() const {
  N child = 2 ∗ rank + 2;
  if (child ⩾ (∗owner_pointer).size()) {
    return R();
  }
  return R(owner_pointer, child);
}

R parent() const {
  if (rank == 0) {
    return R();
  }
  return R(owner_pointer, (rank − 1) / 2);
}

void slide(R& neighbour) {
  swap(neighbour);
}

void swap(R& other) {
  ∗∗this = std::move(∗other);
  std::swap(rank, other.rank);
}

bool operator==(R const& other) const {
  return (rank == other.rank) and (owner_pointer == other.owner_pointer);
}
```

Figure 8. Part of the rank-navigator class template; $\check{\mathcal{V}}$, $\mathcal{R}$, and $\mathbb{N}$ are type aliases for the value type, rank-navigator type, and size type; `owner_pointer` and `rank` specify the owner and the rank of the cell referred to

I warned you of headaches: **const** correctness may be one of the causes. In many cases, it was necessary to provide two overloaded functions: one for immutable objects and another for mutable objects. The problem was severe for the `rank_navigator` class template since we need two classes defining the behaviour of both. There are several alternative ways to avoid code duplication and to define just one class template covering both cases (see, e.g. [1]); we relied on the tools for manipulating types available at the C++ standard library (see, the header <`type_traits`>). As seen from the class

diagram, it should be possible to compare a mutable navigator to a non-mutable one; thus, there are four versions of **operator**==. One detail not shown in the class diagram is the problem with copy constructors and copy assignments, since only three of the four versions should be supported: it should not be possible to convert an immutable navigator (of type $\overline{\mathcal{R}}$) to a mutable navigator (of type $\mathcal{R}$). To solve this, we only provide two of the operations: one from $\mathcal{R}$ to $\mathcal{R}$ (in $\mathcal{R}$), another from $\mathcal{R}$ to $\overline{\mathcal{R}}$ (in $\overline{\mathcal{R}}$), and let the compiler generate the one from $\overline{\mathcal{R}}$ to $\overline{\mathcal{R}}$ (in $\overline{\mathcal{R}}$) automatically if needed. For the details of the template manipulation applied, consult the source code.

Modularity also caused trouble. In the implicit-tree class, the two versions of **operator**[] are private so, since navigators need them, the two navigator classes must be friends of the implicit-tree class. Only an implicit tree should have the permission to construct navigators so that a user does not have any change of destroying the linkage between cells. To guarantee this, in the rank-navigator classes the parameterized constructor is made private and the implicit-tree class (i.e. the type of the owner specified via the template argument) is made their friend. Also, a mutable navigator must have access to the private data of the non-mutable counterpart, and vice versa, so these two classes must be friends of each other.

Circular dependency on template arguments was yet another cause of trouble. For increased flexibility, the type of a navigator could be given as an argument for the implicit tree, but at the same time the navigator should know the type of its owner. That is, how to write the following two lines:

using $\mathcal{R}$ = `cphstl::rank_navigator`<$\mathcal{T}$>;
using $\mathcal{T}$ = `cphstl::implicit_nearly_complete_binary_tree`<$\mathcal{V}$, $\mathcal{S}$, $\mathcal{R}$>;

By some googling, you can see that I am not the only person having this problem. For the purpose of this essay, as an acceptable workaround, the implicit-tree class template is fixed to rely on the rank navigators only.

Having the tree class template available, it is straightforward to write the binary-heap class template. Basically, we can start with a textbook solution and replace array indexing with navigator operations. To give a taste of the implementation, the operations `push` and `pop` are described in Figure 9. Observe that both helper functions `siftup` and `siftdown` take a navigator as reference parameter. This guarantees the correctness provided that the navigator follows the position of the hole, even if it does not refer to the original position. In `pop`, it is permissible to contract the last leaf first after `siftdown`; it will never go to this leaf because the value there cannot be of higher priority than the value used as a substitute (i.e. the same value).

Only you, my dear reader, can answer the following question:

Q: Will these programs appear in a textbook?

In Figure 10, an example of the use of the binary-heap class template is given. This example shows how the configuration of the class templates can be done. Because of the default values set for the type parameters, the binary-heap class template, as `std::priority_queue`, can be used by just

```
public:

  void push(V const& in) {
    R hole = tree.expand();
    siftup(hole, in);
    *hole = in;
  }

  void pop() {
    if (tree.size() == 1) {
      tree.contract();
    }
    else {
      R hole = tree.root();
      R leaf = tree.last();
      siftdown(hole, *leaf);
      hole.swap(leaf);
      tree.contract();
    }
  }
```

```
private:

  void siftup(R& hole, V const& in) {
    while (hole != tree.root()) {
      R p = hole.parent();
      if (not less(*p, in)) {
        break;
      }
      hole.slide(p);
    }
  }

  void siftdown(R& hole, V const& in) {
    R p = hole.left();
    while (p != none) {
      R q = hole.right();
      if (q != none and less(*p, *q)) {
        p = q;
      }
      if (not less(in, *p)) {
        break;
      }
      hole.slide(p);
      p = hole.left();
    }
  }
```

Figure 9. Implementations of `push` and `pop` in the binary-heap class template; $\mathcal{V}$ and $\mathcal{R}$ are type aliases for the value type and navigator type; `tree` and `less` specify the container and comparator in use

```
#include "cphstl/binary_heap.h++"
#include "cphstl/implicit_nearly_complete_binary_tree.h++"
#include "cphstl/resizable_array.h++"
#include <functional> // std::less
#include <memory> // std::allocator

int main () {
  using V = int;
  using C = std::less<V>;
  using A = std::allocator<V>;
  using S = cphstl::resizable_array<V, A>;
  using T = cphstl::implicit_nearly_complete_binary_tree<V, S>;
  using H = cphstl::binary_heap<V, T, C>;

  V input[] = {3, 6, 8, 1, 5, 9, 4, 2, 10, 7, 11, 14};

  H heap;
  for (V x : input) {
    heap.push(x);
  }
  while (heap.size() != 0) {
    heap.pop();
  }
  return 0;
}
```

Figure 10. Simple use case of the binary-heap class template

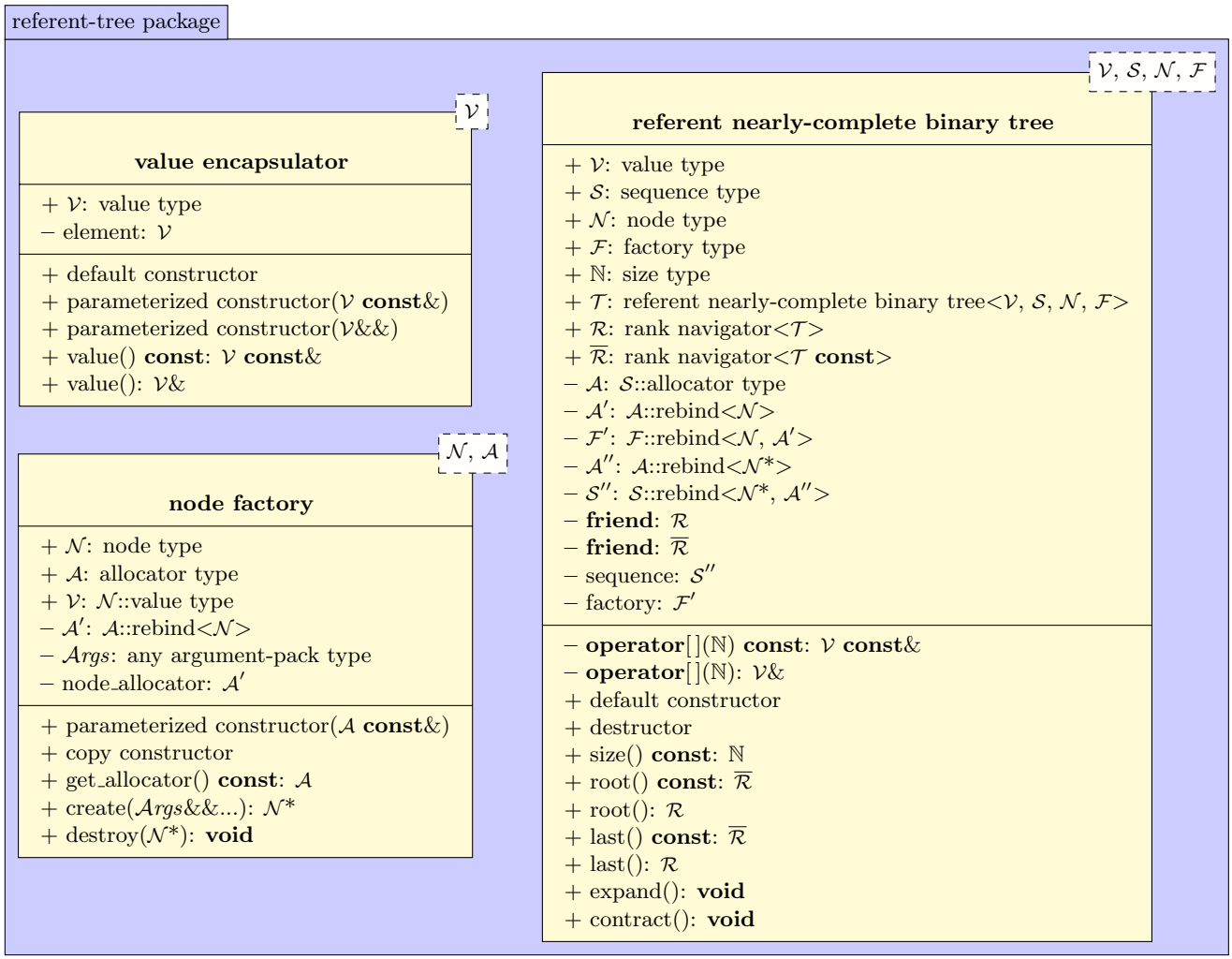

Figure 11. Class templates written for the referent-tree package

specifying the value type. The extra flexibility comes from the fact that the tree type and the comparator type can be customized, too.

5.2 Referent structure

In the referent-tree approach the sequence stores pointers to the values. This is useful in applications where the values are in several containers at the same time. When implementing this approach, it was necessary to write a new tree structure `referent_nearly_complete_binary_tree`. To make its use possible in the mentioned context, the manipulation of values is done outside the class. For this, two more class templates were written: `value_encapsulator` can be used to encapsulate a value, nothing else, and `node_factory` to create and

```
value_encapsulator()
  : element() {
}

value_encapsulator(V const& v)
  : element(v) {
}

value_encapsulator(V&& v)
  : element(std::move(v)) {
}

V const& value() const {
  return element;
}

V& value() {
  return element;
}
```

```
node_factory(A const& a = A())
  : node_allocator(a) {
}

template <typename... Args>
N* create(Args&&... args) {
  N* p = node_allocator.allocate(1);
  try {
    new (p) N(std::forward<Args>(args)...);
  }
  catch (...) {
    node_allocator.deallocate(p, 1);
    throw;
  }
  return p;
}

void destroy(N* p) {
  (*p).~N();
  node_allocator.deallocate(p, 1);
}
```

Figure 12. Central parts of the `value_encapsulator` (left) and `node_factory` (right) class templates; $\mathcal{V}$ and $\mathcal{N}$ are type aliases for the value type and node type; `element` and `node_allocator` are the encapsulated data members

destroy any kind of nodes. No changes were necessary in the rank-navigator class template or in the binary-heap class template. The UML structure diagram in Figure 11 gives an overview of the new class templates.

As seen from Figure 12, the code for both the `value_encapsulator` and `node_factory` class templates is straightforward. The encapsulator provides three constructors, a get and a set function to read and write the stored value. The factory class template provides a parameterized constructor, where an allocator is given, and the two functions use the rebound allocator to allocate space for a node and construct the node in that place (`create`), and to destroy the contents of a node and deallocate the space reserved for that node (`destroy`). To reuse the code for nodes whose constructor accepts more parameters, `create` takes a parameter pack and forwards the arguments as such to the constructor. For correctness, it is essential that the constructor behaves cracefully and does not leave any partially constructed objects behind if it fails. That is, it must be exception safe. If the contructor throws an exception, `create` ensures that no memory is leaked.

Although some changes had to be done to the tree class template, the public interface of `referent_nearly_complete_binary_tree` is still a superset of that of `implicit_nearly_complete_binary_tree`. As can be seen from the class diagram (Figure 11), some type rebindings are necessary in the private part of the class definition, but this has no effect on its use. An interested reader may again peek the source code to see how the template manipulation is done. In essense, the changes are concentrated in the operations **operator**[],

```
V& operator[](N i) {
  return (*sequence[i]).value();
}

R expand() {
  E* p = factory.create();
  sequence.push_back(p);
  return last();
}

void contract() {
  N ℓ = size() − 1;
  E* p = sequence[ℓ];
  factory.destroy(p);
  sequence.pop_back();
}
```

Figure 13. Kernel of the referent-tree class template; $\mathcal{V}$, $\mathbb{N}$, $\mathcal{R}$, and $\mathcal{E}$ are type aliases for the value type, size type, rank-navigator type, and encapsulator type; `sequence` is the container of pointers and `factory` the node factory in use

```
using V = int;
using C = std::less<V>;
using A = std::allocator<V>;
using S = cphstl::resizable_array<V, A>;
using E = cphstl::value_encapsulator<V>;
using F = cphstl::node_factory<E, A>;
using T = cphstl::referent_nearly_complete_binary_tree<V, S, E, F>;
using H = cphstl::binary_heap<V, T, C>;
```

Figure 14. Configuration the binary-heap class template using the referent-tree class template

`expand`, and `contract`, but, as seen from Figure 13, the changes are not big.

With these class templates, the binary-heap class template can be used without modifications. A configuration example is given in Figure 14. The algorithms used in a binary heap have not changed, but the underlying tree structure is different. By separating the representation from the algorithms, the beauty of the algorithms is not disturbed.

5.3 Inverse structure

In the inverse-tree approach the conception, how an array is used, is turned upside down. Nodes are different, navigators are different, and the tree structure is different from those used earlier. Luckily, none of the new class templates is involved. An overview of the developed templates is given in the UML structure diagram in Figure 15.

An inverse-tree node encapsulates a value and a rank, and provides get and set functions for both in a way that is idiomatic for C++. The generic

Figure 15. Class templates written for the inverse-tree package

factory for creating and destroying nodes also works for these new nodes, so that code can be reused. Even though the navigator and tree class templates have to be rewritten, by keeping their public interface unchanged, the binary-heap class template can be reused as well. Let us next consider the new class templates in more detail.

We call navigators that encapsulate a pointer to a node ***link navigators***. Since we need other link navigators for purely linked structures, we call the navigators used here ***inverse link navigators***. In this case, the position is represented by a pair of pointers, one pointing to the owner of a node and another to the node itself. With the owner pointer we can access the sequence and use the ranks stored at the nodes to get from a node to its neighbours. To see the difference between a rank navigator and an inverse link navigator, compare the corresponding operations for the two in Figure 8 (on page 16) and Figure 16.

The inverse tree is very similar to the referent tree, but the navigators are different, so adjustments were necessary in some of the operations. Figure 17 describes the implementations of the central operations of the `inverse_nearly_complete_binary_tree` class template. As show in Figure 18, the use of the developed templates is quite similar to the use of the templates in the earlier packages.

5.4 Linked structure

To sum up our discussion so far, the elements of a binary heap are

- a dynamic set of cells, each storing a value,
- a set of structural invariants guaranteeing that the cells form a nearly-complete binary tree,
- a set of navigation rules showing how to get from a cell to some other cells,
- a set of ordering invariants guaranteeing that the values stored at the cells are in heap order, and
- a set of transformation rules showing how to reestablish the invariants after a modification.

There are three forms of transformations: dynamization transformations like `expand` and `contract`, structural transformations like `slide` and `swap`, and heap-ordering transformations like `siftup` and `siftdown`. In our design, the tree class is responsible for dynamization, the navigator class is resposible for navigation and structural changes, and the binary-heap class is responsible for heap order.

In a linked structure, nodes are the cells and the connections between the nodes are hard-wired using pointers. An overview of the linked-tree package is given in the form of the UML structure diagram in Figure 19. Since the node-factory and binary-heap class templates can be reused, those are not described in the diagram. Compared to the earlier representations, in a linked representation some flexibility is lost, since the interconnections between the components—node, tree, and navigator—are tighter. Contrary

```
Ṽ& operator∗() const {
  return (∗p).value();
}

L left() const {
  N rank = 2 ∗ (∗p).rank() + 1;
  if (rank ⩾ (∗owner_pointer).size()) {
    return L();
  }
  return L(owner_pointer, (∗owner_pointer)[rank]);
}

L right() const {
  N rank = 2 ∗ (∗p).rank() + 2;
  if (rank ⩾ (∗owner_pointer).size()) {
    return L();
  }
  return L(owner_pointer, (∗owner_pointer)[rank]);
}

L parent() const {
  if ((∗p).rank() == 0) {
    return L();
  }
  N rank = ((∗p).rank() − 1) / 2;
  return L(owner_pointer, (∗owner_pointer)[rank]);
}

void slide(L const& neighbour) {
  swap(neighbour);
}

void swap(L const& other) {
  N i = (∗p).rank();
  N j = (∗other.p).rank();
  (∗p).rank() = j;
  (∗other.p).rank() = i;
  (∗owner_pointer)[i] = other.p;
  (∗owner_pointer)[j] = p;
}

bool operator==(L const& other) const {
  return p == other.p;
}
```

Figure 16. Part of the inverse-link-navigator class template; $\check{\mathcal{V}}$, $\mathcal{L}$, and $\mathbb{N}$ are type aliases for the value type, link-navigator type, and size type; `p` specifies the current position and `owner_pointer` is used for navigation

```
N*& operator[](N i) {
  return sequence[i];
}

L root() {
  return L(this, sequence[0]);
}

L last() {
  return L(this, sequence[size() - 1]);
}

L expand() {
  N past = size();
  N* p = factory.create(std::move(V()), past);
  sequence.push_back(p);
  return L(this, p);
}

void contract() {
  N l = size() - 1;
  N* p = sequence[l];
  factory.destroy(p);
  sequence.pop_back();
}
```

Figure 17. Kernel of the inverse-tree class template; $\mathcal{N}$, $\mathbb{N}$, $\mathcal{L}$, and $\mathcal{V}$ are type aliases for the node type, size type, link-navigator type, and value type; `sequence` is the container of pointers to nodes `factory` the node factory in use

```
using V = int;
using C = std::less<V>;
using A = std::allocator<V>;
using S = cphstl::resizable_array<V, A>;
using N = cphstl::inverse_tree_node<V>;
using F = cphstl::node_factory<N, A>;
using T = cphstl::inverse_nearly_complete_binary_tree<V, S, N, F>;
using H = cphstl::binary_heap<V, T, C>;
```

Figure 18. Configuration the binary-heap class template using the inverse-tree class template

to arrays, one can only get from the current node to those nodes for which there is a direct linkage. For example, if a node only stores pointers to its left child and right child, one cannot access the parent, not at least easily.

In fact, I implemented two forms of nodes, one that relies on the standard three-pointer scheme—left child, right child, and parent—and another that relies on a non-standard two-pointer scheme. The pointers are named `first`, `second`, and `third`; a compact node has only the first two of these pointers. A normal node stores a value and the pointers; the dummy nodes—root access point and leaf access point—store only the pointers. The challenge was to implement these node class templates such that both of them could be used

Figure 19. Class templates written for the linked-tree package

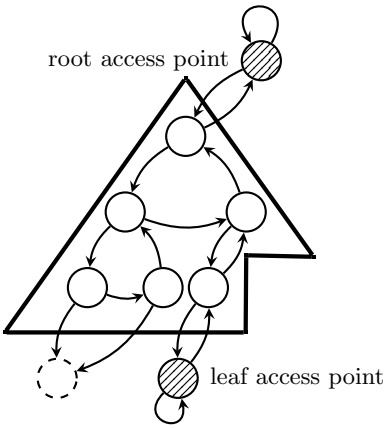

Figure 20. Two-pointers-per-node scheme for a nearly-complete binary tree of size six

by the link-navigator and linked nearly-complete binary-tree class templates at the same time. So this became an exercise in generic programming.

Since the two-pointer scheme is more interesting, let us concentrate on it. An example in Figure 20 illustrates how the two pointers are used. In general terms, a node has a pointer to its left child, which has a pointer to its sibling, and it has again a pointer back to the start node. In more precise terms, to cover the special cases, the two pointers are used as follows. For a normal node, the `first` pointer points to its left child, but for a leaf it points to `none` and for the last leaf it points to the leaf access point. For a right child and for the last leaf, even if it was a left child, the `second` pointer points to the parent; otherwise, it points to the sibling of that node. For the root, the `second` pointer points to the root access point. As to the dummy nodes, the `first` pointer of the root access point points to the root and the `second` pointer points to itself. Finally, the `first` pointer of the leaf access point points to itself and the `second` pointer points to the last leaf.

The node class template provides the functions `left`, `right`, and `parent` to get a pointer to the left child, right child, and parent of a node (see Figure 21 on the left). As seen, they follow closely the given definitions. To update the pointer values in a node, the set functions are more sophisticated. For example, there is no direct way to set the parent pointer, but the parent pointers are kept ajour by the set functions `set_left` and `set_right` (see Figure 21 on the right). Moreover, when the set functions are used, a strict protocol should be followed: A left child should be set before its right sibling. That is, when a right child gets a value different from **nullptr**, its left sibling should not be equal to **nullptr**. For a nearly-complete binary tree, there is no problem in following this protocol, but the compact binary-tree node class template cannot be used in the implementation of a general binary tree where a node can have a right child, but not a left child.

```
N* left() const {
  return first;
}

N* right() const {
  bool no_right = (first == nullptr)
      or (first → second == this);
  if (no_right) {
    return nullptr;
  }
  return first → second;
}

N* parent() const {
  bool on_cycle = (second → second →
       first == this);
  if (on_cycle) {
    return second → second;
  }
  return second;
}
```

```
void set_left(N* q) {
  first = q;
  if (q ≠ nullptr) {
    (*q).second = this;
  }
}

void set_right(N* q) {
  if (q == nullptr) {
    if (first ≠ nullptr) {
      (*first).second = this;
    }
    return;
  }
  (*first).second = q;
  (*q).second = this;
}
```

Figure 21. Get functions (left) and set functions (right) in the compact binary-tree-node class template; $\mathcal{N}$ is a type alias for the node type

```
L left() const {
  N* below =(*p).left();
  bool at_leaf = (below == nullptr) or ((*below).left() == below);
  if (at_leaf) {
    return L();
  }
  return L(below);
}

L right() const {
  return L((*p).right());
}

L parent() const {
  N* above =(*p).parent();
  bool at_root = ((*above).parent() == above);
  if (at_root) {
    return L();
  }
  return L(above);
}
```

Figure 22. Navigation operations in the link-navigator class template; $\mathcal{L}$ and $\mathcal{N}$ are type aliases for the link-navigator type and node type; `p` is the encapsulated pointer to a node

As other navigator templates, a link-navigator class template has the owner as its type parameter, but inside the class only a pointer to a node is used to encapsulate a position. The primary purpose of the link-navigator class template is restrict the access to the pointers in nodes such that sliding and swapping are the only modifying operations allowed. Its secondary purpose is to hide the existence of the dummy nodes from the user. Otherwise, the `left`, `right`, and `parent` functions just rely on the corresponding functions provided by the node class (see Figure 22).

In principle, `slide` is straightforward; it swaps two neighbouring nodes in a linked tree, i.e. the present node should take the place of its neighbour and vice versa. A potential problem with such a swap is that it may also require a change in an access point. However, since the dummy nodes are kept outside the data structure, they will participate a swap only indirectly as a neigbouring node. The key feature of our design is that, at any given point of time, there are at most two other nodes that point to a node and these nodes can be easily accessed from that node via a few pointers. Because `swap` is not as performance critical as `slide` and because they are so similar, the description of `swap` is omitted.

Assume that we want to swap two neighbouring nodes pointed to by `p` and `q`, respectively. First, there are two cases depending on which of the two nodes is the parent of the other. Second, in both cases, there are four subcases depending on whether the two nodes are left or right children of their parents. All the cases are similar, so let us only consider one of them. Without loss of generality, assume that `p` points to the parent of the node referred to by `q` and that both nodes are left children of their parents. The drawing in Figure 23 describes the configurations that are possible depending on the type of the triangle—full, deteriorated, or none—that appear above the node pointed to by `p`, between the two nodes being swapped, and below the node pointed to by `q`. The code in the figure gives the details of the pointer updates made. To simplify the scrutiny, the variable names used in the code of `left_left_case` and the drawing are the same.

By looking at the code, we can make the following observations. First, as an artefact of the framework, some extra pointer updates are made. For the three-pointer scheme, 12 pointer updates are performed although only 10 of them are necessary. For the two-pointer scheme, up to nine pointer updates may be done, but two of them are redundant (see Figure 23 on the right). In total, in these schemes, 19 and 16 pointer assignments are done per `slide`, respectively. Second, the two-pointer scheme performs many more conditional branches than the three-pointer scheme. In addition to the three branches executed before going to the subcases, the three-pointer scheme performs one conditional branch in each `set_left` and `set_right`. This means six conditional branches; i.e. 9 branches per `slide` in the worst case. For the two-pointer scheme, both `right` and `parent` may execute one conditional branch, `set_left` one, and `set_right` two. In total, this means up to 17 conditional branches per `slide`. Since, in the worst-case, both `push` and `pop` perform `slide` once at each level of a heap, this overhead can be considerable.

public:

```
void slide(L const& neighbour) {
  N* q = neighbour.p;
  N* o = (*p).parent();
  N* r = (*q).parent();
  bool p_on_left = ((*o).left() == p);
  bool q_on_left = ((*r).left() == q);
  if (o == q) {
    if (q_on_left) {
      if (p_on_left) {
        left_left_case(r, q, p);
      }
      else {
        left_right_case(r, q, p);
      }
    }
    else {
      if (p_on_left) {
        right_left_case(r, q, p);
      }
      else {
        right_right_case(r, q, p);
      }
    }
  }
  else {
    if (p_on_left) {
      if (q_on_left) {
        left_left_case(o, p, q);
      }
      else {
        left_right_case(o, p, q);
      }
    }
    else {
      if (q_on_left) {
        right_left_case(o, p, q);
      }
      else {
        right_right_case(o, p, q);
      }
    }
  }
}
```

private:

```
// swap p and q, o: parent of p
void left_left_case(N* o, N* p, N* q) {
  N* r = (*o).right();
  N* s = (*p).right();
  N* t = (*q).left();
  N* u = (*q).right();
  (*o).set_left(q);
  (*o).set_right(r);
  (*q).set_left(p);
  (*q).set_right(s);
  (*p).set_left(t);
  (*p).set_right(u);
}
```

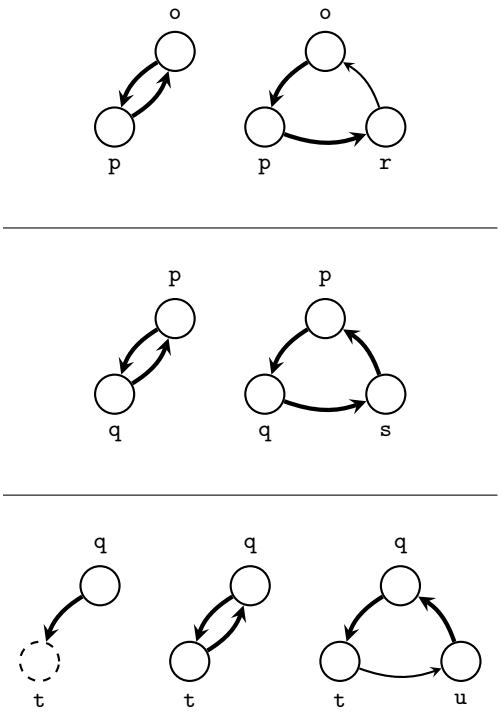

Figure 23. Sliding operation in the link-navigator class template; $\mathcal{L}$ and $\mathcal{N}$ are type aliases for the link-navigator type and node type; `p` and `q` refer to the two neighbouring nodes to be swapped; `left_right_case`, `right_left_case`, and `right_right_case` are similar to `left_left_case`; in the drawing, the pointers that must be updated are thickened

```
L expand() {
  N* q; // parent of the new last leaf
  N* p = factory.create();
  n += 1;
  int const h = __builtin_ctzl(n);
  N* const l = (*leaf_access).parent();
  if (__builtin_popcountl(n) == 1) {
    q = root_access;
    if (n == 1) {
      (*q).set_left(p);
      (*q).set_right(q);
    }
    else {
      for (int i = 0; i != h; ++i) {
        q = (*q).left();
      }
      (*l).set_left(nullptr);
      (*q).set_left(p);
      (*q).set_right(nullptr);
    }
  }
  else if ((n bitand 1) == 0) {
    q = l;
    for (int i = 0; i != h + 1; ++i) {
      q = (*q).parent();
    }
    q = (*q).right();
    for (int i = 1; i < h; ++i) {
      q = (*q).left();
    }
    (*l).set_left(nullptr);
    (*q).set_left(p);
    (*q).set_right(nullptr);
  }
  else {
    q = (*l).parent();
    (*l).set_left(nullptr);
    (*q).set_right(p);
  }
  (*p).set_left(leaf_access);
  (*p).set_right(nullptr);
  return L(p);
}
```

```
void contract() {
  N* q; // new last leaf
  int const h = __builtin_ctzl(n);
  N* const l = (*leaf_access).parent();
  if (__builtin_popcountl(n) == 1) {
    if (n == 1) {
      q = root_access;
    }
    else {
      q = (*root_access).left();
      for (int i = 1; i != h; ++i) {
        q = (*q).right();
      }
      N* p = (*l).parent();
      (*p).set_left(nullptr);
    }
  }
  else if ((n bitand 1) == 0) {
    q = l;
    for (int i = 0; i != h + 1; ++i) {
      q = (*q).parent();
    }
    q = (*q).left();
    for (int i = 0; i != h; ++i) {
      q = (*q).right();
    }
    N* p = (*l).parent();
    (*p).set_left(nullptr);
  }
  else {
    N* p = (*l).parent();
    q = (*p).left();
    (*p).set_right(nullptr);
  }
  (*q).set_left(leaf_access);
  (*q).set_right(nullptr);
  factory.destroy(l);
  n -= 1;
}
```

Figure 24. Dynamization operations in the linked-tree class template; $\mathcal{L}$ and $\mathcal{N}$ are type aliases for the link-navigator type and node type; `n`, `root_access`, `leaf_access`, and `factory` are the data members; `__builtin_ctzl` returns the number of trailing 0 bits and `__builtin_popcountl` the number of 1 bits in the binary representation of an integer (**unsigned long**)

```
using V = int;
using C = std::less<V>;
using A = std::allocator<V>;
using N = cphstl::compact_binary_tree_node<V>;
using F = cphstl::node_factory<N, A>;
using T = cphstl::linked_nearly_complete_binary_tree<V, N, F>;
using H = cphstl::binary_heap<V, T, C>;
```

Figure 25. Configuration the binary-heap class template using the linked-tree class template

In the dynamization of the linked tree, we use the finger-search approach when searching for the parent of the new last leaf in `expand` and the new last leaf in `contract`. To guide the search, we rely on the magical connection between the binary representation of n and the path from the root to the last leaf. By reading the binary representation of n from the most significant bit to the least significant bit, the most significant 1 bit means the root, and in the remaining bit string each 0 means "proceed to the left child" and each 1 means "proceed to the right child". If the binary representation of n has h trailing 1 bits, in `expand` the parent of the new last leaf can be found by going h levels up along the right spine of a subtree, by jumping to the sibling at level h, and by coming down $h-1$ levels along the left spine of the subtree rooted by that sibling. In `contract`, in order to find the new last leaf after removing the current one, the process is the opposite and we have to consider the trailing 0 bits in the binary representation of n. Unfortunately, these recipes do not work when n is $2^h - 1$ in `expand` or 2^h in `contract`, for some non-negative integer h. In these cases, we can find the required node by starting the search from the root and following the left or the right spine of the whole tree. The programs implementing these algorithms are described in Figure 24.

Compared to the other structures, we can conclude that the linked structure is more involved. Partially, this is due to our desire to use less space than the inverse structure. As shown in Figure 25, the developed package can be used in the same way as the earlier packages.

6. How well?

To see how well different binary-heap implementations perform, I made some sanity checks on my computer. Since all code developed is publicly available, in the case of doubt, I encourage the reader to run own experiments in his or her environment.

6.1 Test set-up

I run the experiments under Linux and I compiled the code using `g++`. During experimentation, all unnecessary system services were shut down. The hardware and software specifications of my computer were as follows.
processor: Intel® Core™ i5-2520M CPU @ 2.50GHz × 4
word size: 64 bits
L_1 instruction cache: 32 KB
L_1 data cache: 32 KB
L_2 cache: 262 KB
L_3 cache: 3.1 MB
main memory: 3.8 GB
operating system: Ubuntu 14.04 LTS
Linux kernel: 3.13.0-62-generic

micro-benchmark

```
int const cycle = 74565404;
int const t = 4294967295;

int sum(int* data, int* request, int t){
  int result = 0;
  for (int i = 0; i ≠ t; ++i) {
    result += data[request[i % cycle]];
  }
  return result;
}
```

sequential access

```
for (int i = 0; i ≠ cycle; ++i) {
  request[i] = i % n;
}
```

jumping access

```
int const step = 617;
...
int j = 0;
for (int i = 0; i ≠ cycle; ++i) {
  j += step;
  j = j % n;
  request[i] = j;
}
```

random access

```
for (int i = 0; i ≠ cycle; ++i) {
  request[i] = rand() % n;
}
```

Figure 26. Micro-benchmark used for testing memory performance

compiler: `g++` version 4.8.4
compiler options: `-O3 -std=c++11 -Wall -DNDEBUG`
profiler: `valgrind` version 3.10.0
profiler options: `--tool=callgrind --instr-atstart=no --dump-instr=yes --collect-jumps=yes`
memory allocator: `std::allocator`

Three simple drivers were written which could be used to do the following experiments:

make$_n$**:** Call the constructor for a sequence of size n; the input sequence was specified by an rvalue reference to a sequence of type `std::vector` which made move construction possible, if applicable.

push^n**:** Execute n `push` operations repeatedly.

pop^n**:** Execute n `pop` operations after building a heap with n `push` operations. Only the cost of the `pop` operations was measured.

For each driver, the input could be selected to be

- an increasing sequence of integers $\langle 0, 1, \ldots, n-1 \rangle$ or
- a random permutation of integers $\{0, 1, \ldots, n-1\}$.

Although the drivers were instrumented to accept any value, sequence, and comparator type, all tests used 4-byte **int** values, `std::less` in value comparisons, increasing input in `push`^{n} experiments, and random input in `make`$_n$ and `pop`^{n} experiments.

6.2 Micro-benchmark

In order to understand the consequences of memory hierarchy for the experimental results, I executed a micro-benchmark which accessed an array of n integers (4 bytes each) in different order. I consirered three types of memory

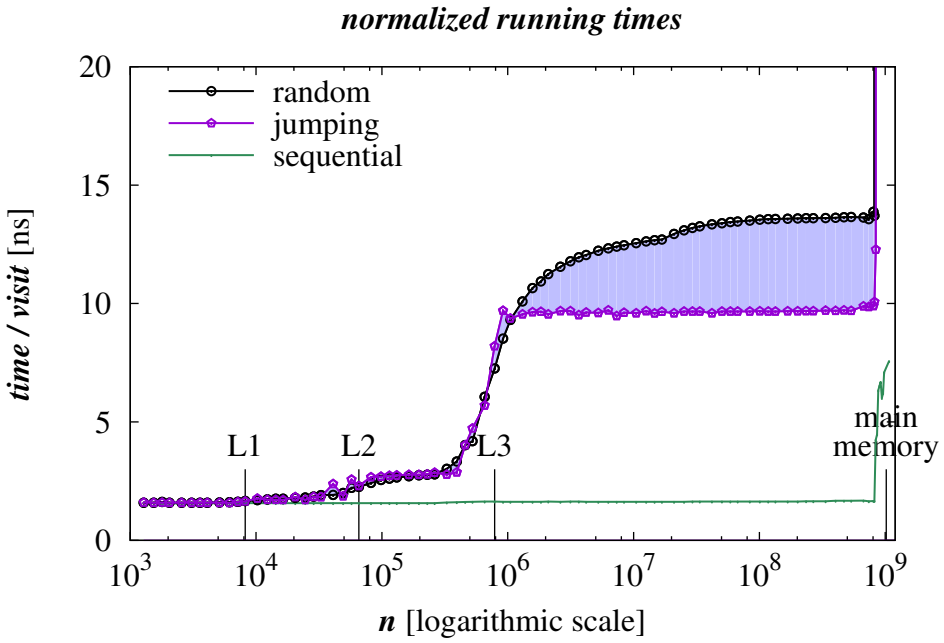

Figure 27. Results of the micro-benchmark

accesses: ***sequential access***, ***jumping access***, and ***random access***. For each of the request sequences, the execution time of a function summing the values was measured. The details, how the request sequences were generated and what the micro-benchmark did, are described in Figure 26.

The results obtained are given in Figure 27; the average running time per access is reported for different array sizes. In the figure, the sizes of the different caches are also drawn. Four plateaus, one corresponding to each memory level, are clearly visible in the curves. The array `request` caused some noise to the results, but not much, since this array was accessed sequentially and it was purposely kept "small". The coloured area shows that a random access involves some additional costs compared to a jumping access; this is due to the translation of virtual addresses to physical addresses.

When the size of the array reached that of main memory, the time per random access rapidly increased to 1.7 ms, which is a factor of 100 000 or more higher than the time per random access in main memory. Definitely, one wants to keep the data structures small in order to process the data in main memory. When these micro-benchmarks were run the virtual-memory support was on; in the later experiments no swap space was reserved for the virtual-memory system so we could be sure that the experiments were run in main memory. It happened that an experiment was killed by the system because the space demand became too large.

To avoid giving this kind of plots every time, I used Figure 27 as a guideline and only report running times for four values $n \in \{2^{10}, 2^{15}, 2^{20}, 2^{25}\}$. The rationale is that these values fall on the intervals of input sizes corresponding to each of the four plateaus, assuming that the data structure was in-place and only stored integers. A data structure that uses more memory will be challenged to do memory accesses that are more expensive than those typical for the present plateau.

To avoid any anomalies with input generation and CPU timing, each test was repeated $t = \max\{2, 2^{25}/n\}$ times. For some data structures, for the largest instance, only one repetition was done to avoid "out-of-memory" signal. If a test required $T(n, t)$ time, I always report the normalized time $T(n, t)/(n \times t)$, i.e. the average running time per value or operation.

6.3 Relative performance

In the first round of experiments, I wanted to test the relative performance of the different approaches. The goal was to provide some guidelines for the users of the framework:

- How much slower is a worst-case-efficient data structure compared to a structure that performs well in the amortized sense?
- How much slower is a space-optimized structure compared to an unoptimized one?
- How expensive should the value moves be before it is worth to use the referent, inverse, or linked structures?
- How expensive is it to provide referential integrity?

All array-based structures were tested with four different dynamic-array implementations: (1) `std::vector`, which is know to provide good performance in practice; and the (2) resizable array, (3) pile, and (4) sliced array taken from the CPH STL [19], which are known to have good worst-case performance. The test results are reported in Table 2, Table 3, Table 4, and Table 5; each table covers one of the developed structures. Recall that the theoretical properties of these structures are listed in Table 1 (on page 12).

As confirmed by the micro-benchmark, random accesses can be considerably slower than sequential accesses. Indirection means random accesses. In this light, there are no big surprises in the results obtained. Albeit, the following points are worthy of attention:

- An implicit structure is an order of magnitude faster than the other three alternatives. Use one of the other structures only if there is a good reason for that. One such reason could be expensive value moves. Even in this case one should probably experiment whether the referent structure leads to a better performance.
- Memory freeing makes `pop` for the implicit structure a factor of two slower. If this is not an important property, it is better to use `std::vector` that does not free the allocated space. On the other hand, in an application that is supposed to run forever, one should consider using some other dynamic array.

Table 2. Average running time [ns] per n for the archetype experiments; ***structure***: binary heap; ***tree: implicit*** nearly-complete binary tree; ***optimization:*** none

n	*std::vector*			***resizable array***			***pile***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	13	20	53	23	27	81	35	53	120	27	48	133
$\mathbf{2^{15}}$	19	33	100	24	41	133	35	100	194	31	73	215
$\mathbf{2^{20}}$	20	45	156	25	56	206	35	112	329	31	101	336
$\mathbf{2^{25}}$	20	51	321	25	76	388	34	166	817	31	129	687

Table 3. Average running time [ns] per n for the archetype experiments; ***structure***: binary heap; ***tree: referent*** nearly-complete binary tree; ***optimization:*** none

n	*std::vector*			***resizable array***			***pile***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	48	53	85	59	62	122	71	90	160	61	81	159
$\mathbf{2^{15}}$	54	67	153	61	75	207	71	140	311	66	106	275
$\mathbf{2^{20}}$	54	80	457	59	101	518	71	158	958	67	139	670
$\mathbf{2^{25}}$	56	89	970	60	110	1 026	70	211	2 130	67	169	1 351

Table 4. Average running time [ns] per n for the archetype experiments; ***structure***: binary heap; ***tree: inverse*** nearly-complete binary tree; ***optimization:*** none

n	*std::vector*			***resizable array***			***pile***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	51	78	111	56	94	123	69	126	157	58	109	133
$\mathbf{2^{15}}$	54	105	262	57	130	285	71	183	349	60	157	290
$\mathbf{2^{20}}$	57	160	1 038	57	200	1 291	71	268	1 399	62	238	1 372
$\mathbf{2^{25}}$	57	207	2 534	59	258	3 061	70	323	3 199	62	292	3 485

Table 5. Average running time [ns] per n for the archetype experiments; ***structure***: binary heap; ***tree: linked*** nearly-complete binary tree; ***optimization:*** none

n	***three pointers***			***two pointers***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	63	132	156	66	187	182
$\mathbf{2^{15}}$	70	180	300	74	257	341
$\mathbf{2^{20}}$	73	228	1 102	77	353	1 453
$\mathbf{2^{25}}$	73 [a]	270 [a]	2 517 [a]	76	445	3 335

[a] Only one repetition; for two repetitions the structures took too much space and the experiment was killed by the system

– When the problem instances reach the size of main memory, it is important to use a space-efficient data structure. In such situation, the use of a space-consuming resizable array and a three-pointer linked structure cannot be recommended.
– For all structures, `pop` is an order of magnitude slower than `push`. In the case of repeated `push` operations, most of the data on a path from a new leaf to the root are cached. Actually, in this use case the cache behaviour is asymptotically optimal, provided that the size of the cache is reasonable [5]. In the case of random `pop` operations, it is probably that, at the lower levels of the heap, the visited data are not cached which means cache misses.
– For the inverse structure, the type of dynamic array did not play a significant role. Random memory accesses dominate the overall costs so the details in the array implementation seem to be less important.
– A pile uses more memory than a sliced array and it was slower in almost all experiments; if it was faster, it was only marginally so. Therefore, I left it out of the subsequent experiments.
– A linked structure can be competitive to an inverse structure, but the overhead caused by node swapping is clearly visible.

6.4 Performance of the competitors

To ensure that the framework does not have any obvious shortcomings, in the second round of experiments, it was natural to consider how well do the competitors perform in the same test set-up. The following binary-heap implementations were selected for this comparison.

– `std::priority_queue` shipped with `g++` . Be aware that both `std::make_heap` and `std::pop_heap`—used by the constructors and `pop`, respectively—implemented `siftdown` in a bottom-up manner [22, Exercise 5.2.3–18] by going to a leaf following the dominant path of children and finding the correct place of the substitute on the way back up along the same path.
– Williams' original array-based implementation converted from Algol to template-based C++, but still using **goto**s.
– Jensen's referent binary-heap implementation taken from the CPH STL; it uses the same `std::make_heap`, `std::push_heap`, and `std::pop_heap` functions as `std::priority_queue`. In the original version, the sequence storing the references was fixed to be of type `std::vector`. Using the rebinding tricks from Section 5.2, the data structure was extended to employ other types of dynamic arrays as well.
– The single-heap framework that relied on the inverse approach and swapped nodes as recommended in [10, Section 6.5]. This program was written by Edelkamp and Katajainen [18], and used in two earlier experimental studies [8, 12]. The top-down binary-heap heapifier follows closely the guidelines given by Floyd [15].

Table 6. Average running time [ns] per n for the archetype experiments; ***structure***: `std::priority_queue` shipped with `g++`

n	*std::vector*			***resizable array***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	4	21	39	16 [a]	54	101	21 [b]	62	127
$\mathbf{2^{15}}$	7	30	69	20 [a]	77	151	24 [b]	96	197
$\mathbf{2^{20}}$	9	42	146	21 [a]	104	231	24 [b]	126	297
$\mathbf{2^{25}}$	10	52	557	21 [a]	139	395	25 [b]	153	596

[a] Construction from resizable array, not from `std::vector`

[b] Construction from sliced array, not from `std::vector`

Table 7. Average running time [ns] per n for the archetype experiments; ***structure***: ***array-based*** binary heap programmed by Williams [36]

n	*std::vector*			***resizable array***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	4	14	37	10	24	55	14	31	86
$\mathbf{2^{15}}$	11	19	77	17	34	92	22	47	144
$\mathbf{2^{20}}$	11	25	124	17	49	176	22	63	261
$\mathbf{2^{25}}$	11	31	283	17	56	622	22	79	869

Table 8. Average running time [ns] per n for the archetype experiments; ***structure***: ***referent*** binary heap programmed by Jensen

n	*std::vector*			***resizable array***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	42	53	76	55	109	118	57	108	146
$\mathbf{2^{15}}$	48	62	196	58	142	261	61	144	259
$\mathbf{2^{20}}$	71	80	1 024	80	191	1 270	85	200	956
$\mathbf{2^{25}}$	75	92	2 661	83	228	2 993	86	239	2 273

Table 9. Average running time [ns] per n for the archetype experiments; ***structure***: ***inverse*** binary heap programmed by Edelkamp and Katajainen [18]

n	*std::vector*			***resizable array***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	*	64	85	*	80	128	*	86	144
$\mathbf{2^{15}}$	*	76	173	*	97	239	*	116	260
$\mathbf{2^{20}}$	*	98	709	*	128	967	*	156	1 173
$\mathbf{2^{25}}$	*	116	1 985	*	153	2 334	*	189	3 089

* Non-trivial constructor was not provided

Table 10. Average running time [ns] per n for the archetype experiments; ***structure***: purely ***linked*** binary heap proposed by Goodrich et al. [16]

n	*three pointers*		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	49	57	77
$\mathbf{2^{15}}$	53	68	147
$\mathbf{2^{20}}$	59	82	607
$\mathbf{2^{25}}$	60 [a]	94 [a]	1 415 [a]

[a] Only one repetition

– A purely pointer-based implementation as described in the book by Goodrich et al. [16]. Finger search was done according to the guidelines given in [16, Exercise C-7.9]. This version swaps the values inside the nodes, not the nodes themselves.

For an overview of the implementation of `push` in these programs, see Figure 3 (on page 7).

The experimental results for these competitors are reported in Table 6, Table 7, Table 8, Table 9, and Table 10, respectively. When discussing the results, I consider the four approaches one at the time.

Of the array-based solutions (see Table 2, Table 6, and Table 7), Williams' original program was a clear winner. However, it constructs a heap by repeated insertions, which can lead to $\Theta(n \lg n)$ worst-case running time; the test case triggered its average-case behaviour which is $O(n)$ [17]. On the other hand, as the $\texttt{push}^n$ experiment shows, this is not necessarily a disaster since the cache behaviour is asymptotically optimal. The $\texttt{make}_n$ experiment reveals that heap construction is not done well enough in our framework. The reason is that Floyd's [15] heap-construction algorithm sets additional requirements for the nearly-complete binary tree; it should be possible to access the nodes in reverse breadth-first order which works fine for an array, but not for a tree. In the tested version, a heap was built recursively, though its worst-case behaviour was still linear (see, e.g. [16, Section 7.3.5]).

In the $\texttt{push}^n$ experiment, Williams' original was faster than the other two. After some profiling, I found two reasons why `std::priority_queue` was so slow. First, it relies on fast iterator operations. For the more advanced dynamic arrays, these operations are not always as fast as the corresponding operations for pointers. Second, in array indexing, `std::priority_queue` used signed integers, whereas the other two used unsigned integers. By running the instruction-cost micro-benchmark from Bentley's book [3, Appendix 3], I could verify that, in my computer, division was a bit slower for signed integers than for unsigned integers. Even though the same arithmetic operations were executed, the difference in types could explain the difference in overall performance.

From the results of the $\texttt{pop}^n$ experiment, two observations can be made. First, for `std::priority_queue`, for the largest instance, a resizable array gave

better results than `std::vector`. This is an indication that occasional global rebuildings can be harmful since some useful information will be flushed out from the caches. In a resizable array, rebuildings are done incrementally, so only part of a cache will be reserved for the rebuilding process. Second, our framework is again seen to have some overhead compared to Williams' original. One reason is that Williams made sure that the last leaf always had a sibling so he could avoid one branch in the inner loop of `siftdown`.

The results for the referent structures (see Table 3 and Table 8) were surprising since the specialized program was sometimes slower than the framework. Especially, for the largest problem instance, Jensen's `pop` was a factor of 1.5–3 slower. In principle, this function just called `std::pop_heap`. Profiling showed that, in a profiling session for $n = 2^{20}$ using `std::vector`, the following six functions used almost 90 % of all clock cycles:

- **operator**+ for iterators 28.2 %
- copy constructor for iterators 14.5 %
- **operator**∗ for iterators 10.3 %
- `std::__adjust_heap` (`siftdown`) 24.3 %
- `less_ref` 6.5 %
- `std::less` 4.7 %

Compared to our implementation (see Table 13), the relative cost of navigators was about the same as that of iterators, the relative cost of the heap and tree together was around 28 %, and the relative cost of value comparisons was a bit smaller, but this could not explain the difference. The results are understandable if the corresponding operations were cheaper for our implementation.

Of the two frameworks relying on the inverse approach (see Table 4 and Table 9), the one written by Edelkamp and Katajainen was clearly superior. Seems that my fear that the earlier framework was not good enough was unjustified. But a sliced array makes the heap operations slower and a purely linked structure can challenge the inverse structure. The two-pointer version uses less space than the inverse version even with a sliced array and sometimes the three-pointer version was faster. The framework used in the earlier experimental studies is more general than the one presented in this essay. Probably, by specializing it somewhat, it could be improved. Based on the profiler data, it was difficult to see how to improve the framework presented here since the costs were split across the classes.

When comparing the results for the implicit approach (see Table 2, Table 6, and Table 7) and those for the linked approach using value swaps (see Table 10), it must be concluded that the latter does not have a niche in the market. On the one hand, it was slower than array-based solutions, and, with a few exceptions, this was true even for the worst-case ones. On the another hand, it cannot guarantee referential integrity. However, the results give us a deeper understanding of the costs involved in node swapping. For integer data, a value swap is definitely a cheaper alternative to a node swap requiring 16 pointer assignments and up to 17 conditional branches.

6.5 Fine-tuning the framework

In the third round of experiments, I wanted to understand what was behind the obtained results and how to improve them if possible. It was time to write some experimental code and do some more profiling. In all profiling experiments, in the array-based implementations I used `std::vector` and in the linked alternative I used the three-pointer version. Before seeing what the profiler will tell us, I recommend that you stop reading the essay for a while and think what would be your answer to the following question.

Q: Where are the performance bottlenecks in the programs under consideration?

All four versions run exactly the same constructor, `push`, and `pop` function. As the first approximation, the bottleneck of `push` must be the inner loop of `siftup`, and the bottleneck of both constructor and `pop` must be the inner loop of `siftdown`. Thus, overall performance seems to depend on the efficiency of the small functions called inside these inner loops.

To get an exact answer, I used a profiler to collect the frequency how often individual functions were called and what were their share of the total cycle consumption. In all profiling sessions, the problem size n was set to 2^{20} and the number of repetitions t to 4. Instrumentation was programatically enabled before making the constructor, `push`, and `pop` calls, and disabled after these calls. Table 11, Table 12, and Table 13 summarize the information produced by the profiler for the $\texttt{make}_n$, $\texttt{push}^n$, and $\texttt{pop}^n$ experiments, respectively. The problem with the generated data was that it was detailed. Therefore, the summary tables list the costs per class by collecting together the costs incurred by, or indirectly associated with, the member functions of each class. Observe that the sum of the individual costs is less than 100 % because of rounding and because some work done outside could not be directly associated with any of the classes.

According to the 80/20 law, 80 % of the execution time of a program is spent executing 20 % of the code. I will leave it for you to decide if this law is valid for this framework. Often, three to four classes dominate the overall costs, but depending on the way the framework is used, these classes vary. However, it is clear that the navigators are in the centre of the performance equation. If they could be improved, the overall performance will improve.

Let us stop talking and go to the business. Recall our to-do list:

Q: How could heap construction, `push`, and `pop` be improved?

Next I will explain what I did. The code is for your eyes only; a library user should not know much about the optimization details. In fact, I am not proud of all the hacks used, but you can peek the source code if you are interested. Here I will only give the general idea behind each optimization.

(1) I applied Williams' optimization in `siftdown` so that it was always called for odd n. Hereafter it was not necessary to consider the special case—either inside or outside the inner loop—whether a cell has one child or

Table 11. The costs [%] associated with different classes; $n = 2^{20}$; ***experiment***: `make`^{n}; ***optimization:*** none

class	***implicit*** `std::vector`	***referent*** `std::vector`	***inverse*** `std::vector`	***linked three pointers***
heap	9.4	7.0	8.6	9.6
tree	12.4	11.2	9.6	8.5
navigator	41.7	33.6	30.5	31.2
sequence	17.0	11.4	10.7	0.9
iterator	9.2	7.2	7.3	8.1
node	–	3.3	6.3	19.8
factory	–	2.8	3.3	1.5
allocator	6.9	6.9	21.4	18.1
comparator	2.0	1.6	1.6	1.8

Table 12. The costs [%] associated with different classes; $n = 2^{20}$; ***experiment***: `push`^{n}; ***optimization:*** none

class	***implicit*** `std::vector`	***referent*** `std::vector`	***inverse*** `std::vector`	***linked three pointers***
heap	13.7	12.2	12.4	10.6
tree	14.6	15.3	14.3	5.1
navigator	56.0	50.4	40.3	40.2
sequence	10.3	8.7	11.7	–
node	–	4.6	12.3	36.4
factory	–	0.2	0.6	0.2
allocator	1.2	4.3	4.4	2.8
comparator	3.5	3.1	3.2	2.7

Table 13. The costs [%] associated with different classes; $n = 2^{20}$; ***experiment***: `pop`^{n}; ***optimization:*** none

class	***implicit*** `std::vector`	***referent*** `std::vector`	***inverse*** `std::vector`	***linked three pointers***
heap	14.0	12.8	14.0	14.2
tree	13.9	15.3	12.0	1.4
navigator	52.8	48.3	45.0	47.3
sequence	14.3	13.3	12.7	–
node	–	4.7	10.2	31.2
factory	–	0.1	0.1	0.1
allocator	0.3	1.3	1.4	1.0
comparator	4.3	4.0	4.3	4.4

not. In heap construction, a heap of size n or $n-1$ was constructed, depending on which one was odd, and the last value, if any, was pushed into the heap afterwards. In `pop`, if n was even before the call, the last value was detached from the tree and used as a substitute for the value removed from the root. On the other hand, if n was odd before the call, the substitute was kept in its place and the last leaf was removed from the tree first after `siftdown`.

To facilitate this change, the tree structures were extended with three new operations: `detach` which cuts off the last cell from the tree and returns its name, `attach` which moves a detached cell back to the tree, and `remove` which removes a detached cell from the custody of a data structure. No fancy data structures were maintained to keep track of the cells outside a data structure, but still in its custody; it was the responsibility of the user to ensure that no memory was leaked. Furthermore, for the structures that cannot guarantee referential integrity, the maximum number of detached cells was limited to one. Hence, the last cell could just be hidden from the heap if necessary.

When a hole was swapped with a detaiched cell, it was no more necessary to support a general `swap` operation for cells, but a simpler `replace` operation could be used, which replaced a hole with a detached cell and moved the hole outside instead. For the linked version, about half of the pointer updates were saved this way.

(2) I extended the tree classes to provide a constructor that constrcuted a tree of populated cells in one go, instead of repeatedly adding holes and letting the heap-building procedure populate and build the heap. To make this extension possible, the sequence class had to allow move/copy construction from a given sequence and a function `reserve`. For the implicit structure this reduced the expected number of value moves performed from $\sim 4.74n$ to $\sim 1.58n$ (measured experimentally). Even after this change, in the node-based structures, the nodes were still created one by one in order to be able to destroy them.

(3) I removed recursion from the heap-construction procedure. Starting with a populated tree of values, the iterative procedure made a post-order traversal of the tree and called `siftdown` at each branch node after both of its subtrees contained a heap. The traversal was done by maintaining a bit stack in a computer word telling whether the nodes visited above the current node are left or right children. Logical bitwise operators were used when manipulating the bit stack. Thus, heap contruction was done using $O(1)$ words of additional memory.

(4) I extended the navigators to provide two additional operations: `is_root` and `is_leaf`. Then I insisted that the other programs should follow a strict protocol and call `left` and `right` only if a node is not a leaf, and `parent` only if the node is not a root. Hereafter all the **if**-outside checks could be removed from the navigation functions.

(5) As to the semantics of `slide` and `replace`, I made an addition that, after the operation, the reference to the other node is undefined and cannot

be used. This was only relevant for the rank navigators which could not retain referential integrity. Hereafter the call of `std::swap` inside `slide` and `replace` could be replaced with an assignment. In some cases, this swap used more than 10 % of the total running time.

(6) I made the linked structure fully symmetric by introducing two more dummy nodes, a sibling of the last leaf and a sibling of the leaf-access point. Hereafter the triangles were perfect, except the topmost triangle involving the root-access point, but this did not require any special handling. Due to Williams' optimization, the dummy for the last leaf did not cause any harmful interference in `pop` because this dummy was not in use when the number of nodes was odd. To utilize the symmetry, I added a few new functions to the node classes with the aim that `slide` and `replace` would do exactly the required pointer updates and that conditional branches could be avoided as far as possible. The two set functions were complemented with four other set functions that operated with triangles (`set_top_corner`, `set_left_corner`, `set_right_corner`, and `make_triangle`) and two reset functions (`reset_left` and `reset_right`) that reset a pointer to **nullptr**. After the redesign, in the two-pointer case, at most 14 pointer assignments and four conditional branches were executed per `slide`. Three of the conditional branches were needed to determine in which case we are and one to check whether the bottommost triangle exists or not. Using conditional moves and an array of function pointers, the number of conditional branches could be reduced from four to one, and the remaining branch was easy to predict, but this branch optimization did not pay off.

I expected that these optimizations would give a normal code-tuning improvement in the running time (up to 20 % and, if I am lucky, more). By performing an additional round of experiments I could confirm that code tuning was effective, but, as expected, it did not help much in cases where the memory performance was the bottleneck (compare the results in Table 14, Table 15, Table 16, and Table 17 to those reported earlier in Table 2, Table 3, Table 4, and Table 5). After these optimizations the performance of the programs generated by the framework could match, if not exactly at least almost, that of their competitors. In some cases, the generated programs were significantly faster.

The only unpleasant feature of these optimizations was that they made the use of the framework more complicated. At least for me, the debugging sessions tended to become longer since at some points I had not followed the protocols set out for the optimized functions.

7. What if?

In this section you are allowed to ask questions, also such that I cannot answer; an answer may require some further research.

Let us start with some questions that I can answer.

Table 14. Average running time [ns] per n for the archetype experiments; ***structure***: binary heap; ***tree: implicit*** nearly-complete binary tree; ***optimization: tuned***

n	*std::vector*			***resizable array***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	5	15	33	9	26	64	17	40	104
$\mathbf{2^{15}}$	10	22	75	16	40	115	23	59	172
$\mathbf{2^{20}}$	10	29	123	16	56	179	24	78	267
$\mathbf{2^{25}}$	10	36	278	15	69	344	23	108	559

Table 15. Average running time [ns] per n for the archetype experiments; ***structure***: binary heap; ***tree: referent*** nearly-complete binary tree; ***optimization: tuned***

n	*std::vector*			***resizable array***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	40	48	66	45	62	104	51	71	135
$\mathbf{2^{15}}$	48	53	132	54	76	183	58	93	236
$\mathbf{2^{20}}$	48	62	431	54	92	512	60	114	603
$\mathbf{2^{25}}$	47	72	1 023	54	108	1 002	60	145	1 290

Table 16. Average running time [ns] per n for the archetype experiments; ***structure***: binary heap; ***tree: inverse*** nearly-complete binary tree; ***optimization: tuned***

n	*std::vector*			***resizable array***			***sliced array***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	52	77	98	53	82	127	64	97	142
$\mathbf{2^{15}}$	57	104	254	59	113	285	69	138	296
$\mathbf{2^{20}}$	57	143	1 012	60	162	1 219	72	190	1 314
$\mathbf{2^{25}}$	56	173	2 658	60	204	2 847	72	246	3 285

Table 17. Average running time [ns] per n for the archetype experiments; ***structure***: binary heap; ***tree: linked*** nearly-complete binary tree; ***optimization: tuned***

n	***three pointers***			***two pointers***		
	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$	$\texttt{make}_n$	$\texttt{push}^n$	$\texttt{pop}^n$
$\mathbf{2^{10}}$	55	117	108	59	144	124
$\mathbf{2^{15}}$	63	158	221	67	194	262
$\mathbf{2^{20}}$	67	192	963	70	254	1 291
$\mathbf{2^{25}}$	68 [a]	223 [a]	2 286 [a]	69	337	2 957

[a] Only one repetition

Q: Which kind of changes would be necessary if support for `extract` was required?

Conceptually, this would require that the binary heap provided locators (navigators without any navigation abilities); in our current implementation navigators were private. Then `insert` should return a locator to the inserted cell, which can be used by `extract` to release that cell. The actual implementation of `extract` is not difficult: Replace the cell being removed by the last cell (not a value), and call both `siftdown` and `siftup`, starting from the new location of the replaced cell. This would fix the heap order independent of whether the priority was increased or decreased at the place of the replacement. If you want to see some real code, inspect, for example, the implementation of `extract` in the single-heap framework in [18].

Q: You say locators, should there be support for iterators too?

In the implementation of `expand` and `contract` for a nearly-complete binary tree, we saw that it is not difficult to find a successor or a predecessor of a cell. Therefore, there is no problem in supporting **operator**++ and **operator**−− for a locator. However, these are not necessarily constant-time operations. Also, **operator**∗ should not be allowed to modify the cell contents, because this would give an opportunity for the user to break the invariants maintained.

Q: At some point you mentioned `merge`; was this meant seriously?

For the linked representation, two binary heaps of size m and n can be merged in $O(\lg m \times \lg n)$ worst-case time [27]. However, I have never seen this algorithm implemented. I did not give it a try since, if I really needed a mergeable priority queue, I would use some of the competitors, e.g. many versions of binomial queues [34] are readily available at the CPH STL.

Q: Since there are many different representations of a nearly-complete binary tree, at some point it might be necessary to convert one representation to another. Where should these conversions be implemented?

It would be natural to provide for any nearly-complete binary tree, say of type $\mathcal{T}$, two types of copy constructors, one creating a new copy from a tree of type $\mathcal{T}$ and another template-based version creating a new copy from a tree of type $\mathcal{X}$, where the concept requirements for $\mathcal{X}$ and $\mathcal{T}$ are the same. In order to implement this idea, in the tree class templates the allocator types must be made public and the class templates must provide `get_allocator` function as standard-library containers.

Standard-library containers do not have constructors from containers or ranges, only from iterator pairs, but the standard library could be extended to provide these more general constructors, too. Now, for example, the constructor of `std::priority_queue` relying on a resizable array only allows a construction from a resizable array, not from a `std::vector`. When trying the construction via iterator pairs, the implementation turned out to come with a new set of concept requirements. For example, the dynamic arrays should

provide general `insert` accepting iterator pairs, i.e. some kind of multiple `push_back` at one go, but such operation was not available in the dynamic arrays taken from the CPH STL.

Q: What if you used multiary heaps instead of binary heaps? Are they not faster?

After the publication of the paper by Ladner and LaMarca [24], many people argue that multiary heaps are faster than binary heaps. Sanders [28] showed—and my experiments confirm his observation—that an engineered version of a binary heap, e.g. the tuned version discussed in this essay, is equally fast as, or faster than, a 4-ary heap; 8-ary or 16-ary heaps are already slower. We have seen that cache behaviour really matters. If it is important in your application, you should study Sanders' work before writing your own cache-efficient priority queue.

Q: [Stefan Edelkamp] In the work you mentioned, Sanders uses an insertion buffer to improve `push` and an extraction buffer to improve `pop`. Does buffering make sense in your framework too?

Yes, two layers will do better even without Sanders' fancy second layer. For binary heaps, buffering was proposed by Wegner and Teuhola [35] and, according to the experiments carried out by Bojesen [4], it worked well when the size of the buckets was set to $M/4$, M being the size of the largest cache in bytes (L_3 cache in my computer). In my experiments, for the largest instance ($n = 2^{25}$), the running time of `pop` almost halved when buffering was used compared to Williams' original program, and for other instances, no slowdown was experienced. To use this idea in the framework, it would be necessary to write yet another version of the binary-heap class template. One should also be aware that after this change the running times of `push` and `pop` are again amortized, not worst-case.

I know that you expect me to come outside my comfort zone; there must be some wilder questions to pose.

Q: By Table 1, when you asked for four bytes from `std::allocator`, you got 32 bytes. Does this mean that I should use a custom allocator in every performance critical application?

If you cannot afford this type of waste, you have to. In the experiments I have done, a pool allocator has not improved the runtime performance, so it is necessary to use a custom allocator only if the space usage becomes a problem. Observe also that, when I talk about the amount of space used, I mean the amount of allocated space. I do not consider memory fragmentation. If all allocated blocks are of 32 bytes or larger, with high probability, the amount of wasted space due to external fragmentation will be small because a released block can be used for other purposes as well.

Another aspect related to memory management is memory layout. In another study [14], we played with different memory layouts for a nearly-

complete binary tree. It turned out that, for large problem instances, a cache-friendly layout could speed up access operations by a factor of two. On the other hand, for small problem instances, the performance slowdown was significant because the navigation within the tree required more CPU time. One of the design goals of the C++ standard library was to separate memory management from data structures. Unfortunately, the cache-friendly layout considered was highly dependent on the data structure so this separation was not achieved.

Q: How do you think frameworks will change the marketplace?

I see frameworks everywhere; they should be more visible in textbooks and software libraries. A textbook on data structures should present all data structures in the form of customizable frameworks, simply because the one-size-fits-all approach does not work in software production. Some books have tried this using the object-oriented approach, but, in my opinion, this has been a failure. At the moment, I believe in the template-based approach, but it may be that I am too optimitic about its success. And a software library on data structures should provide customizable frameworks, instead of offering one implementation per data structure. But also here I can be wrong because of the usability issues involved.

8. Afterword

To make the implementation of the framework manageable, between the cell structure and the operations provided for the user, I used a small set of primitives as middleware which facilitated navigation within, dynamization of, and transformation of the cell structure. Because of this middleware, the algorithms used when implementing the high-level operations could be kept unchanged. That is, by separating the representation from the operations, the beauty of the algorithms could be retained.

The foundation for the framework described, implemented, and benchmarked was laid down in the textbook of Goodrich et al. [16]. The basic idea was solid, but (1) their implementation contained minor inconsistencies [16, Section 7.3.3] (e.g. a pointer-based implementation does not need any initial capacity), (2) the underlying tree implementation had some extra fat [16, Section 6.4.2] (e.g., as pointed out by the authors, the external nodes need not be stored explicitly), (3) the crucial details of the pointer-based solution were left for the exercises [16, Section 7.5], and (4) the proposed class template was only dry run. In this essay I filled in the gaps.

There are two things that I would like to see improved in most textbooks on data structures and algorithms:

(1) Instead of just describing how to implement a dynamic array that has good amortized behaviour (as, e.g. in [10, Section 17.4]), one should describe an implementation that supports the fundamental operations (`operator[]`, `push_back`, and `pop_back`) in $O(1)$ worst-case time as space efficiently as possible (e.g. describing different versions of piles [20]).

(2) Instead of assuming that **new** (`malloc` in C) and **delete** (`free` in C) are magically available, describe how these primitives can be implemented in $O(1)$ worst-case time (e.g. using the colouring algorithm [25]). Tell also the bad news about memory fragmentation [26, 37].

As should be clear from this essay, questions related to dynamization and memory management can be pivotal for understanding the performance of data structures.

As shown, the efficiency of different variations of a binary heap can vary a lot. However, my experimentation was by no means exhaustive. It is here that crowdsourcing comes into the picture. The programs described, including benchmarking tools, are publicly available, so anyone interested can test how different alternatives work in his or her environment. Based on the experimental results, a serious user can then customize the framework for his or her needs.

I have to warn the reader for not becoming overly enthusiastic about adaptable component frameworks. It is more complicated to implement a framework capable of producing several data structures than to implement a single data structure. Also, maintenance of such frameworks can be challenging. If a change is necessary in one of the implementations or if the framework is to be extended, the developer must understand the consequences of such a change in all the underlying implementations. Masochistic programmers call this fun.

The last point that I want to make is whether John and Jane Doe are ready for using a customizable software library? Maybe the use is too complicated; in particular, if C++ templates are involved. I do not know; you tell me.

Software availability

The programs described, implemented, and benchmarked are available via the home page of the CPH STL (`www.cphstl.dk`) in the form of an electronic appendix and a `tar` file.

Acknowledgements

I thank Max Stenmark for the discussions that helped me to get the final details of the framework in place.

References

[1] M. Austern, Defining iterators and const iterators, *C/C++ User's Journal* **19**, 1 (2001), 74–79.

[2] M. H. Austern, B. Stroustrup, M. Thorup, and J. Wilkinson, Untangling the balancing and searching of balanced binary search trees, *Software Pract. Exper.* **33**, 13 (2003), 1273—-1298.

[3] J. Bentley, *Programming Pearls*, 2nd Edition, Addison-Wesley, Inc. (2000).

[4] J. Bojesen, Heap implementations and variations, Written Project, Dept. Comput. Sci., Univ. Copenhagen (1998). Available at `http://www.diku.dk/forskning/performance-engineering/Jesper/heaplab/heapsurvey_html/Welcome.html`.
[5] J. Bojesen, J. Katajainen, and M. Spork, Performance engineering case study: Heap construction, *ACM J. Exp. Algorithmics* **5** (2000), Article 15.
[6] G. S. Brodal, A survey on priority queues, *Space-Efficient Data Structures, Streams, and Algorithms*, *LNCS* **8066**, Springer (2013), 150–163.
[7] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick, Resizable arrays in optimal time and space, *WADS 1999*, *LNCS* **1663**, Springer (1999), 37–48.
[8] A. Bruun, S. Edelkamp, J. Katajainen, and J. Rasmussen, Policy-based benchmarking of weak heaps and their relatives, *SEA 2010*, *LNCS* **6049**, Springer (2010), 424–435.
[9] The C++ Standards Committee, Standard for Programming Language C++, Working Draft **N4296**, ISO/IEC (2014).
[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3th Edition, The MIT Press (2009).
[11] Dept. Comput. Sci., Univ. Copenhagen, The CPH STL, Website accessible at `http://www.cphstl.dk/` (2000–2015).
[12] S. Edelkamp, A. Elmasry, and J. Katajainen, The weak-heap family of priority queues in theory and praxis, *CATS 2012*, *Conferences in Research and Practice in Information Technology* **128**, Australian Computer Society, Inc. (2012), 103–112.
[13] A. Elmasry and J. Katajainen, Worst-case optimal priority queues via extended regular counters, E-print **arXiv:1112.0993**, arXiv.org (2011).
[14] A. Elmasry and J. Katajainen, Branchless search programs, *SEA 2013*, *LNCS* **7933**, Springer (2013), 127–138.
[15] R. W. Floyd, Algorithm 245: Treesort 3, *Commun. ACM* **7**, 12 (1964), 701.
[16] M. T. Goodrich, R. Tamassia, and D. M. Mount, *Data Structures and Algorithms in C++*, John Wiley & Sons, Inc. (2004).
[17] R. Hayward and C. McDiarmid, Average case analysis of heap building by repeated insertion, *J. Algorithms* **12**, 1 (1991), 126–153.
[18] J. Katajainen, Priority-queue frameworks: Programs, CPH STL Report **2009-7**, Dept. Comput. Sci., Univ. Copenhagen (2009).
[19] J. Katajainen and A. M. Maniotis, *Dynamic arrays in practice* (2015). Work in progress
[20] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *WAE 2001*, *LNCS* **2141**, Springer (2001), 39–50.
[21] J. Katajainen and B. Simonsen, Adaptable component frameworks: Using `vector` from the C++ standard library as an example, *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, ACM (2009), 13–24.
[22] D. E. Knuth, *Sorting and Searching*, *The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman (1998).
[23] D. Kolb, *Experiential Learning: Experience as the Source of Learning and Development*, Prentice Hall (1984).
[24] A. LaMarca and R. E. Ladner, The influence of caches on the performance of heaps, *ACM J. Exp. Algorithmics* **1** (1996), Article 4.
[25] M. G. Luby, J. S. Naor, and A. Orda, Tight bounds for dynamic storage allocation, *SIAM J. Discrete Math.* **9**, 1 (1996), 155–166.
[26] J. Robson, An estimate of the store size necessary for dynamic storage allocation, *J. ACM* **18**, 3 (1971), 416–423.
[27] J. R. Sack and T. Strothotte, An algorithm for merging heaps, *Acta Inform.* **22**, 2 (1985), 171–186.
[28] P. Sanders, Fast priority queues for cached memory, *ACM J. Exp. Algorithmics* **5** (2000), Artilce 7.
[29] B. Simonsen, A framework for implementing associative containers, CPH STL Report **2009-3**, Dept. Comput. Sci., Univ. Copenhagen (2009).
[30] Stack Exchange Inc., Heap implementation using pointer, Worldwide Web Document

(2013). Available at `http://codereview.stackexchange.com/questions/33365/heap-implementation-using-pointer`.

[31] Stack Exchange Inc., Pointer-based binary heap implementation, Worldwide Web Document (2013–2014). Available at `http://stackoverflow.com/questions/19720438/pointer-based-binary-heap-implementation`.

[32] A. Stepanov, Foreword, *D. R. Musser, G. J. Derge, and A. Saini, STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 2nd Edition, Addison-Wesley (2001), xxi–xxvii.

[33] B. Stroustrup, *The C++ Programming Language*, 4th Edition, Pearson Education, Inc. (2013).

[34] J. Vuillemin, A data structure for manipulating priority queues, *Commun. ACM* **21**, 4 (1978), 309–315.

[35] L. M. Wegner and J. I. Teuhola, The external heapsort, *IEEE Trans. Softw. Eng.* **15**, 7 (1989), 917–925.

[36] J. W. J. Williams, Algorithm 232: Heapsort, *Commun. ACM* **7**, 6 (1964), 347–348.

[37] D. Woodall, The bay restaurant—A linear storage problem, *Amer. Math. Monthly* **81**, 3 (1974), 240–246.