A Comparative Analysis of Three Different Priority Deques

Søren Skov & Jesper Holm Olsen

Abstract. In this project we compare the practical effectiveness of three different algorithms for "priority deques", namely MinMax-heaps, The Deap and Interval Heaps. By implementing the algorithms and running benchmarks we find that interval heaps are the most effective, mainly due to its simplicity and similarity to standard heaps.

We also discover some crucial shortcomings of Svante Carlsson's deap algorithm and propose solutions for these.

Our code is targeted towards submission to the Copenhagen STL project so we implement a "PriorityDeque"-class, in which the programmer can choose the underlying algorithm.

CPH STL Report 2001-14, October 2001

1. Introduction

This report is a "written project" done in the fall of 2001 at the Institute of Computer Science, University of Copenhagen (DIKU). The goal of the project is to implement three different algorithms for a priority deque in order to compare the practical running times and submit the implementations to the Copenhagen-STL project. The workdescription for this project can be found in appendix Appendix A.

We will start by giving a short description of what a priority deque is, as well as some remarks on the C++ standard and how our implementation shall conform to this.

Afterwards we will look at the algorithms and give a short description of how they work. In some cases our implementation differs from the pseudo-code in the articles, and also the article describing the interval heap does not provide pseudo-code. In these cases we will provide pseudo-code that describes our implementation.

As we have located two cases where the deap algorithm described by Carlsson fails we will also describe how we corrected these, and give the pseudo-code for the corrected algorithm.

We will then discuss how to make an appropriate benchmark strategy, and we will give a short description of the expectations we have for the results of the benchmarks. Finally we will analyze the results of the performance benchmarks and discuss which algorithm works best in practice.

In the conclusion in section 7 we will discuss the possibilities for further work with priority deques.

1.1 The Priority Deque Data Structure

A queue is a data container which makes it possible to insert elements, and then retrieve them in the order they where inserted. That is, it is possible to retrieve the element that has been waiting for the longest time from a queue.

In a priority queue it is possible to use other ways of determining which element is next in line. It is possible to associate some sort of ordering, and then retrieve the smallest (or largest) element from the priority queue. A priority queue can be implemented using a heap.

The C++ standard [6] specifies a worst-case running time for the operations on a priority queue (where N is the number of elements) to be:

Operation	Time complexity
Get smallest element	O(1)
Delete smallest	O(log N)
Constructing the structure	O(N)
Insert a new element in the priority deque.	O(log N)

All three priority deque algorithms have the same worst case running time.

As opposed to a priority queue a priority deque holds the possibility to get both the smallest and the largest element using the same structure. This can be used for different purposes: The article describing MinMax[1] mentions applications for external quicksort, and adaptations of the algorithm to provide *constant-time* FindMedian and *logarithmic-time* DeleteMedian, while the article on interval heaps [9] discuss utilization of priority deques for sorting and solving certain computational geometry problems. We will not pursue this further in this report.

1.2 C++ like Interface

As described in the work description, one of the purposes of this project is to add the priority deque algorithms to the Copenhagen STL project[5]. This section will give a short description of which requirements from the C++ standard influences the way we can implement the algorithms.

C++ does not include a definition of priority deques, but instead we will use the requirements for ordinary priority queues, and expand it to fit priority deques. This is easily done, since the structures are very similar.

The only change we have made to the interface defined in [6] is that the `top()` and `pop()` functions has been changed to become `top()`,`bottom()`, `pop_top()` and `pop_bottom()` to allows reading and removing both the smallest and the largest elements from the priority deque. Since the ordering-operator can be change by the users, we use "top" as the name of methods that work with one end, and "bottom" for methods that work with the other. If the standard ordering-operator `less()` is used the "top" of the priority deque will hold the smallest elements, and "bottom" the largest element. In the rest of this report we will use natural words like min, max, smallest and largest and the ordering-operator (<) with the assumption that the strict weak ordering is the operator `less()`.

A priority queue is defined to use three template parameters, the first is used to specify the type of the elements in the priority queue. The second specifies the container the algorithm uses. By specifying this the user can select to use other containers than the default (`vector`). The third specifies the strict week ordering that is used to compare the elements, as mentioned above the default is `less()`.

We have used SGIs[8] implementation of a priority queue[7] as a inspiration for the design and implementation of our priority deques.

We have decided to make all three algorithms available for the users, so we have implemented the "PriorityDequeue" class so that it has a fourth template parameter that specifies the algorithm. After the benchmarking we will find the fastest algorithm and make this the default algorithm. This gives the user the possibility to choose the algorithm, but the default will be the one that performs best in our benchmarks[1].

[1] An example of the use of this can be found in the file `test.cpp` in appendix Appendix B

Besides the interface, STL also defines requirements for the time complexity for the algorithms used (see section 1.1). All three algorithms meets these requirements.

2. The Algorithms

In this section we will give a description o how each of the algorithms work. We will start by describing how a standard priority queue works using a heap and then use this as a reference point when describing the three priority deque algorithms.

The source code that implements the algorithms can be found in appendix Appendix B.

The first priority deque algorithm was the MinMax heap suggested by Atkinson et. al. in 1986 [1]. Svante Carlsson published another algorithm in 1987 called The Deap [4] which was a theoretical improvement of the MinMax-heap . Leeuwen & Wood present a new approach called interval heaps in 1988 which is closely related to normal heaps.

2.1 A Heap Implementation of a Priority Queue

An ordinary implementation of a priority queue is based on a binary tree. The tree is arranged to always maintain the invariants:

Min-ordering The value stored in a node X in the tree is less that or equal[2] to all values stored in the children of X.

Balanced tree The tree is balanced so that all $\lfloor \log_2 N \rfloor$ levels in the tree is complete and that the level $\lfloor \log_2 N \rfloor + 1$ is filled from left to right.

2.2 Min-Max Heaps

Min-Max heaps[1] are based on a normal heap, where the main difference is that the Min-Max heap, has alternating min- and max-levels in the heap. The idea is that nodes in a min-level has the property that the node holds the smallest value in the subtree with the node as root. Correspondingly nodes in max-levels hold the largest of all value in the subtree. This ordering of the values in alternating min and max levels, is illustrated by an example in figure 1. The smallest element in a MinMax heap is located in the root node, and the largest element is in either of the two nodes on the second level.

So the invariant of the Min-Max heap guarantees that the first min-level holds the smallest element and that the first max-level holds the largest element. This makes finding the smallest and largest element rather simple.

The implementation of a Min-Max heap is fairly straightforward, and the pseudo-code in the article is very detailed and precise [1]. As a result we have implemented MinMax-heaps as described in the original article, and

[2] In max-queues it is max-ordering where the relation is greater than or equal.

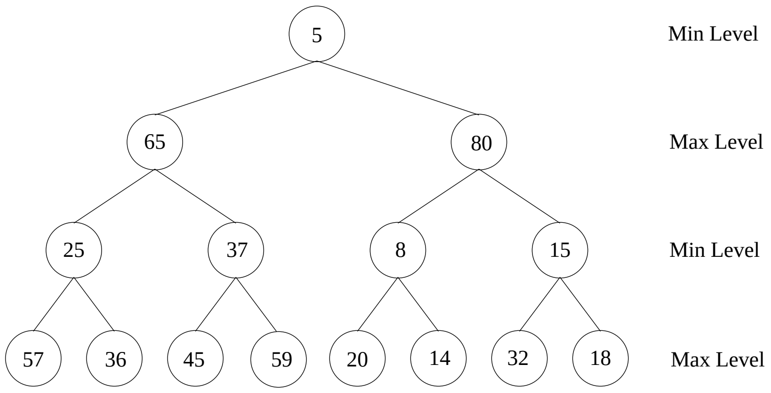

Figure 1. Illustration of a minmax priority dequeue. (Based on figure 1 in [1])

we will therefore not present pseudo-code here, but instead refer to the article and our implementation (see appendix Appendix B).

2.3 The Deap

Svante Carlsson proposes a solution for a priority deque using a *double ended heap*, called a "Deap" [4]. The deap is introduced as a theoretical improvement of the MinMax-heap from the previous section.

A deap has an empty root-node and the minimum and maximum elements are located on the second level, index one and two respectively (see figure 2). The left sub-tree is thus called the "min-heap" and the right sub-tree the "max-heap".

The invariant of this data structure is that any node in the min-heap is min-ordered (as defined in section 2.1) and any node in the max-heap is max-ordered. Also, any leaf node in the min-heap is smaller than the corresponding node in the max-heap. The "corresponding node" of a node in the min-heap is computed the following way:

> *Let a be half the width of the level of node k. Then the corresponding node of k is $k+a$ if it exists and otherwise $(k+a)/2$. The corresponding node of a node in the max-heap is computed as $k-a$.*

The calculation of $(k+a)/2$ is used if the level is not fully occupied, as shown in figure 2 — here the corresponding element for the nodes with values 7 and 12 is not available, and therefore 68 is chosen instead.

2.3.1 Carlsson's Pseudo-Code

In his article (see [4]) Carlsson explains his algorithm by means of pseudo-code, which unfortunately is written rather ambiguously at times and thus is hard to understand, as the following example shows:

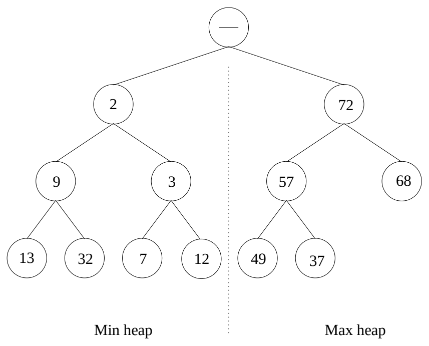

Figure 2. An example of a deap

In the pseudo-code for the creation of a deap there is a *for*-loop described with the text:

> *"**for** all positions — J — with any information about an element with a larger index, starting with the last **do**".*

We have tried to find the exact meaning of this, but have not been able to find a clear interpretation of this. Possible interpretations are:

- For all elements.
- For all elements, except the last
- For all nodes that are not leaf-nodes.

We have determined that the only interpretation that will actually work is the interpretation "For all elements", so we have chosen to use this, but it still shows a remarkable ambiguity in his presentation of the deap.

Turning to his Ph.D-thesis (see [3]) in search of a better explanation, one finds a Pascal-implementation of his algorithm, which unfortunately has errors[3] and does not work when run by hand, which we tried. A well-hidden technical report[4] from the Department of Computer Science at Lund University (see [2]) contains roughly the same code and explanations as in the Ph.D-thesis — along with the same errors, unfortunately. Unless we have completely misunderstood his algorithm, we must conclude that he has not fully covered all aspects of his proposed data-structure. We have identified the special cases which he does not handle and found solutions to deal with them. This will be explained in the remaining parts of this section. First we will show two examples of when the problems occur and propose solutions, and afterwards we discuss what the cause of the problem is.

[3] See [3], p. 69-70. In the procedure "TrickleDownMin" he in some cases uses the "MaxCousin"-procedure as if it returns a value and in other cases as if it returns an index.
[4] We had to get it retrieved from the basement archives at Lunds University.

2.3.2 Our Pseudo-Code

We believe that the reason Carlsson's algorithm fails, is that he presumably forgets some very important special cases for finding the corresponding element, when a level is not fully occupied. This is a problem in two cases:

1. Missing a potential swap with a corresponding element
2. Choosing the wrong corresponding element.

In order to explain this in detail, let us first have a look at the pseudo-code for our implementation. There should be functions for creating the deap from scratch (*create*), for inserting new elements (*insert*) and for removing either the min- or max-element (*DeleteMin* and *DeleteMax*): (Note that the `DeleteMax` is similar to `DeleteMin` and `TrickleUpMax` is similar to `TrickleUpMin` and so these are not shown).

```
BuildHeap(A)
   for (i = last element of A downto 0) do
      if (i is in min-heap)
         DeleteMin(i,i,A[i])
      else
         DeleteMax(i,i,A[i])

Insert(A,X)
   // A = Array containing the Deap
   // X = element to be inserted
   i = index of first free element
   A[i] = X
   if (i is in min-heap)
      InsertMin(i,1,X)
   else
      InsertMax(i,1,X)

InsertMin(i,limit,X)
   // limit = top level for comparison
   if (i is a parent && the corresponding element < X)
      A[i] = A[corresponding]
      TrickleUpMax(corresponding, limit, X)
   else
      TrickleUpMin(index, limit, X)

InsertMax(i, limit, X)
   if (i is a leaf)
      corres = corresponding element
      // Carlsson does not do this
         if (i is the only child of its parent
             && limit < parent(i))
            corres = right child of corres
            if (X < A[corres])
```

```
                A[i] = A[corres]
         // Carlsson does not do this either
            if (the corres element of i is a parent)
               corres = largest child of corres
         if (X < A[corres])
            A[i] = A[corres]
            TrickleUpMin(corres,limit, X)
         else
            TrickleUpMax(i,limit,X)
      else
         TrickleUpMax(index,limit,X)

   DeleteMin(i,index,X)
      // i is the limit from TrickleUp
      // index is the index of element to be deleted
      // X is the value to replace the removed value

      while (index is a parent)
         A[index] = value of smallest child of index
         index = index of smallest child of index
      InsertMin(index,i,X)

   TrickleUpMin(i,limit,X)
      p = the parent of i
      while (i has a parent &&
             the parent is on a level >= limit &&
             the parent is < X)
         A[i] = A[p]
         i = p
         p = parent of i
      A[i] = X
```

Let us explain some details of the above implementation, before we discuss the mentioned problems of the algorithm.

In `InsertMin` and `InsertMax` there appears a parameter called "limit", which in the call from `Insert` is set to the value 1. The purpose of this is to set a limit to how high `TrickleUpMin` and `TrickleUpMax` should continue. This is needed in `create` which works by heapifying small subdeaps repeatedly and therefore should not proceed to the top of the deap. This parameter is also passed to the `DeleteMin` function.

In `TrickleUpMin` we compare the element X with its parent to see if X is smaller and should be swapped. This continues until X is larger than its parent. Carlsson computes this by first doing a binary search in order to find how far the swaps should go (finding the level of the first element smaller than X), before he executes the swaps. Carlsson does this to achieve a theoretical improvement of the running time. We have chosen not to implement this as we do not expect it to improve the running time in practice

— thus we find it smarter to do it our way.

2.3.3 *Missing a potential swap with a corresponding element*

In the the function `InsertMax` there is a remark in the pseudo-code: "Carlsson does not do this", which of course calls for further explanation. This is the first place where — as far as we can see — Carlsson's algorithm fails. Let us demonstrate the problem with an example:

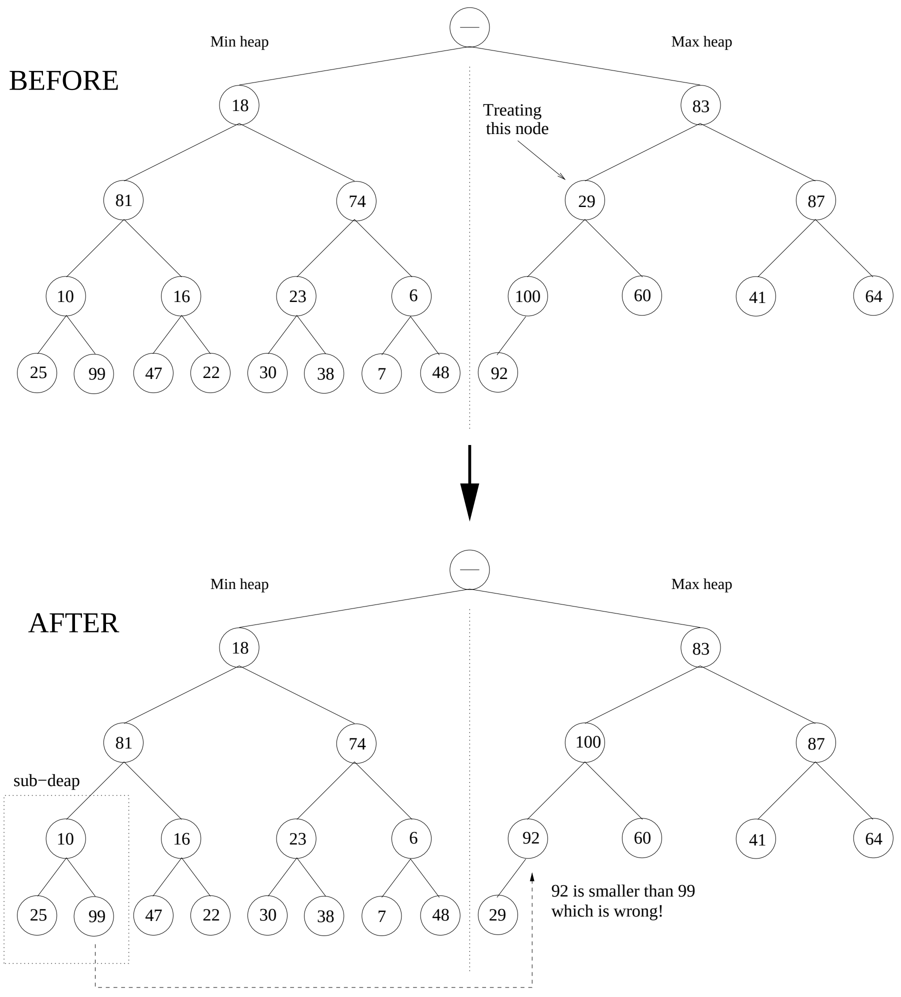

Figure 3. This example shows the error when create is treating node 5 with value 29

.

On figure 3 we are in the progress of creating a deap. This implies treatment of all nodes from the bottom and up. In the "before"-example we have reached the node with index 5 (value 29). What will happen now is that 29 will be extracted and the greatest elements of its children will be "pulled-up". This results in node 11 with value 100 being moved to node 5 and thereafter node 23 with value 92 being moved to node 11. Hereafter `InsertMax` will be called on 29 (at node 23) and in that process 29 will be compared to its corresponding element (node 15 with value 25) which is smaller than 29 and thus no swapping will take place.

The problem here is (as shown in the "after"-example) that node 16 with value 99 is now greater than its corresponding node, which is node 11 with a value of only 92. This breaks the invariant, that any leaf-node in the min-heap is smaller than its corresponding node in the max-heap. This happens in the special case where the last element is the only child (a left child), because checking the corresponding element of the element in current treatment (here 29) is not enough, there is a risk that the parent is also changed. This example shows a situation where the error occurs. The solution is to make an extra comparison of the two nodes (here 92 and 99). This however should only be done once the sub-deap (marked "sub-deap" on the figure) has been heapified, because only hereafter can the heap-property be assumed. This means, that in practice the algorithm should only do this extra comparison, if we are treating a node located at least two levels up in the heap (for example node 5). This can be done with the "limit" value (as described in section 2.3.2).

This implies the extra check, which is shown in our pseudo-code for "InsertMax".

2.3.4 Choosing the wrong corresponding element

Another problem is not choosing the correct corresponding element. This occurs when removing an element from the max-heap, or during create.

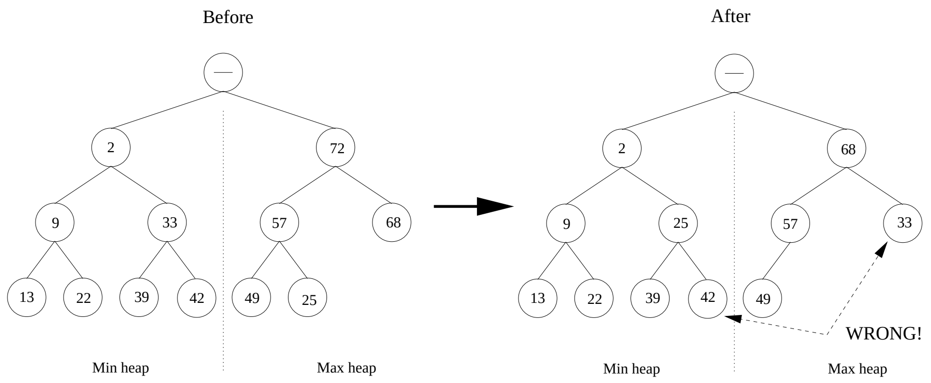

Figure 4. This example shows that Carlsson's algorithm chooses the wrong corresponding element when removing a max-element. The correct corresponding element would have been 42 instead of 33.

In the example on figure 4 we have a deap of size 12 from which we remove the max-element 72. What happens now is that the element with the last index (here index 12, value 25) is moved to the empty node at index

2 and hereafter compared with its children in the max-heap until it reaches a leaf or until its children are all smaller than itself. In this example we just swap with the leaf at index 6 which now gets the value 25. (The same situation could occur during create, if node 2 initially held the value of 25).

In Carlsson's algorithm the node with value 25 would now be compared with its corresponding element (node 4, value 33) and swapped if this is larger, but that is not correct if the corresponding node has any children: Since the corresponding node is in the min-heap it's children will always be greater than itself and thus the invariant that any element in the min-heap is smaller than its corresponding element in the max-heap will be broken. Our solution is that instead of comparing with the corresponding element, we compare with the *largest child* of the corresponding element, thus keeping the invariant. This is done in the pseudo-code at the comment "Carlsson does not do this either".

In the example on figure 4 the node with value 42 should have been swapped with 25 which would then be "Trickled up" until its parent is smaller than itself and end in node 4. This way a leaf-node in the max-heap can have up to three corresponding nodes in the min-heap. This implies the problem that when a new value is inserted into such a node, a comparison will have to be made on all three elements in the min-heap. This is not described in the article and is crucial for the correctness of the algorithm.

2.3.5 Discussion of the Problem

Our analysis of Carlsson's deap algorithm shows that he omits to handle certain special cases when the last level in the deap is not fully occupied. In these situations there are potential problems with choosing the "correct" corresponding element. A bit more formal specification of the problem would be to express the relationship between finding the corresponding element from the min-heap to the max-heap and the other way around:

The problem is that in Carlsson's specification, the relationship is asymmetrical. (That X is corresponding to Y does not necessarily imply that Y is corresponding to X)

i.e. it matters from which point of view you find the corresponding element (from either the min- or max-heap). Our solution is to extend the *one-to-one* relationship with a *one-to-many* relationship between corresponding nodes, as shown on figure 5.

This implies changing the definition of the corresponding element to:

> *Let a be half the width of the level of node k. If a node k is in the min-heap, then the corresponding node of k is $k + a$ if it exists and otherwise $(k + a)/2$. The last situation occurs if the level is not fully occupied. If k is in the max-heap the corresponding elements is the elements in the min-heap that has k as their corresponding element.*

These extra calculations and comparisons will of cause have an impact on the performance of the algorithm: The theoretical implication of the changes is that a number of additional comparisons are performed when

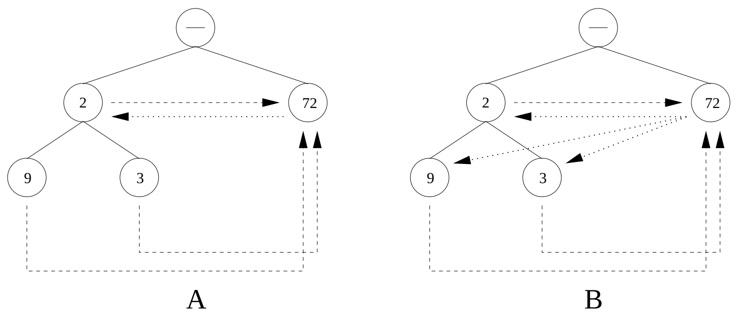

Figure 5. The relationship of corresponding elements in Carlsson's algorithm (case A) and our extension (case B).

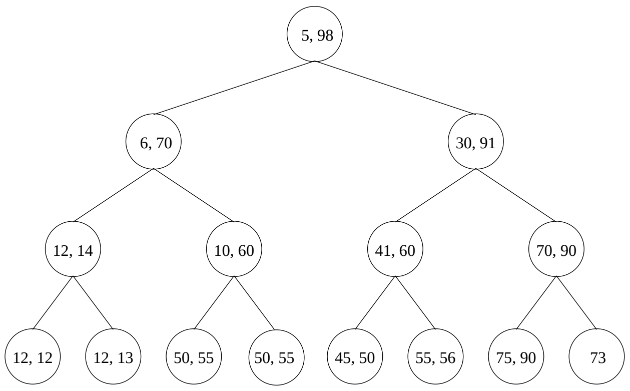

Figure 6. Illustration of a interval heap.

processing leaf nodes. In the worst case there will be two extra comparisons in the methods: Creation the heap, `deleteMin()`, `deleteMax()` and `insert()`.

2.4 Interval Heaps

The basic idea in interval heaps is that each node contains an interval. This is done by letting the node hold two elements: One that serves as the "lower-bound" of the interval and one that is the "upper-bound" of the interval. The invariants of an interval heap are:

A node contains an interval All nodes hold two elements in an interval $[a, b]$, where $a \leq b$. The exception is the last node in the heap that might only have one element a. In this case the interval is considered to be $[a, a]$.

Heap-ordering The intervals of the children E and F of a node X are contained in X's interval. ($E \subset X$ and $F \subset X$)

An example of a interval heap is shown in figure 6.

The article describing interval heaps[9] does not give pseudo-code for the algorithm but describes them in the text. Their reason for this is, that interval heaps are very similar to ordinary heaps and are therefore "trivial" to implement. We will therefore describe our implementation by giving pseudo-code for the procedures. As described in the article the procedures for working with an interval heap are very similar to a standard heap.

In the pseudo-code in this section will use the following assumptions:

- We use the name "element" to describe a pair of numbers that describe an interval, and the word "value" to describe a single number in the heap.
  If the total size of the heap is odd, the last element in the heap will only hold one value. This element is thought of as holding the interval [a,a].
- The function i.first() (respectively i.last()) gives the smallest (respectively largest) value in the interval i.
- For functions that are very similar we will only present the pseudo-code for the "min"-version. The corresponding "max"-version can be constructed by changing of the comparisons and a few other trivial things (see our source code for details in appendix Appendix B).
- A is the container used to hold the heap. We will use syntax that treats A as an array.

The "`create`"-function is passed a set of values as parameter (contained in `A`) and it will rearrange `A` so that it becomes an interval heap.

```
create(A)
   for (i = last element in the heap downto 0)
      if (i.first > i.last)
         swap i.first and i.last
      heapifyMin(i);
      heapifyMax(i);
```

The procedure "`heapifyMin`" establish the ordering of the first() values in the tree with root `i`. The procedure assumes that the sub-trees under the element `i` is already heap-ordered.

```
heapifyMin(i)
   if i has a child
      X = the smallest of i's childrens first();
      j = the element that contains X
      if X < i.first()
         swap X and i.first()
         if the new i.first() > i.last()
            swap those
         heapifyMin(j)
```

"`Push`" adds a value `X` to the interval heap.

```
push(X)
   insert X in the end of A.
   i = the element X was inserted into
   if size of A is odd
```

```
      p = the parent to i
      if X < p.first()
         bubbleUpMin(i)
      else
         bubbleUpMax(i)
   else   // size is even
      if X < i.first
         swap X and i.first
         bubbleUpMin(i)
      else
         bubbleUpMax(i)
```

The procedure "bubbleUpMin" is used to move a value with index i up through the "min"-heap until the min-ordering in the "min"-heap is reestablished.

```
bubbleUpMin(i)
   if i has a parent (p)
      if i.first() < p.first()
         swap them
         bubbleUpMin(p)
```

"deleteMin" deletes the smallest value, and reestablish the interval-ordering.

```
deleteMin()
   A[0] = A[size-1]
   heapifyMin(0)
```

3. Optimizing the Code

This section covers some technical details of the implementation.

3.1 Computation of Logarithm

When computing the expected running time for the algorithms, it is assumed by the articles that it is possible to compute $\log_2$ in constant time. This assumption does not always hold for the build-in functions in C++. Some libraries does not even include a $\log_2$, so it will have to be computed by $\frac{\log_e(X)}{\log_e(2)}$.

We have used a function (ilogb()for computing $\log_2$, that when compiled using gcc, will result in a constant time computation.

We made a small comparison of the complicated computation using $\log_e$, and the constant time $\log_2$. For some of the methods (create and push in deap) this improvement removed approximately 50% of the total time used in the methods.

3.2 Compiler Problems

We started this project using the GNU C++ compiler "gcc version 2.95.2" that is the currently installed version at DIKU. Regrettably we discovered that our implementation of the MinMax algorithms worked without optimization, but stopped giving the correct answers when we added compiler optimization (`-O`). We discovered that exactly *this* version of gcc had a number of bugs in the optimization part. We confirmed that this was the cause of our problems by compiling our source code on Solaris using Suns "CC"-compiler and with an older version of g++ (2.95), both using compiler optimization, without any problems. On this basis we assume that our code is correct.

4. Benchmark Strategy

In this section we will describe our approach for measuring the relative performance of the three algorithms. The main purpose of the performance benchmarks is to be able to compare the three algorithms in order to determine which is fastest in practice.

As mentioned in section 3.2 we discovered a bug in optimization in g++ that influences the MinMax algorithm. We have measured the performance in three different ways: One using Sun's CC compiler on a SparC computer with optimization, one using gcc on an Intel PC without optimization, and one with optimization. All three shows the same ranking of performance for the algorithms. We have then chosen to describe the optimized gcc version even though we know that the MinMax does not give the correct result. We do this since the optimized result is the one that will be of relevance to users of our class[5]. Furthermore gcc claims to have located and corrected the bugs in later versions of gcc, so the problem should hopefully be without relevance soon. We know that using a version of an algorithm that does not give the correct result might give a slightly misleading result for this algorithm, but we are more interested in the result for the other algorithms *optimized* and the relative difference in performance and so this approach seems reasonable.

We find the most interesting results to be the results achieved on "state-of-the-art" hardware, so we will run all the benchmarks on a fairly fast machine. This is a PentiumIII 866 MHz with 16kB level 1 cache, 256kB level 2 cache and 512 MB RAM. The operating system was Linux with kernel 2.2.

We will in all the benchmarks include measurements of the priority queue included in the standard C++ library. When comparing with the standard priority queue one should bear in mind, however, that the comparison is not fair, since the standard heap does not provide the same functionality as a priority deque, and therefore should not be considered "equal". An obvious idea could be to use two heaps to get the min/max-functionality,

[5] If they are interested in performance, they will surly optimize their code.

but besides using twice the space and time there are also some problems in keeping the two heaps "synchronized" (making sure that when removing an element from one heap it can not be extracted from the other) that would require even more work than just two standard heaps and thus increasing the running time.

4.1 Choosing Benchmarks

We will do two kinds of benchmarks: The first is a benchmark for each of the methods in the priority deque, which will show the performance of each method. The second is an approximation to a real-life scenario, where an application will alternate between the different methods.

The first kind results in the following four benchmarks:

1. The method for constructing a priority deque. (The class constructor)
2. The method for inserting a new element in the priority deque. (The method `push()`)
3. The methods for getting and removing the smallest element from the priority deque. (The methods `top()` and `pop_top()`)
4. The methods for getting and removing the largest element from the priority deque. (The methods `bottom()` and `pop_bottom()`)

For the second kind of benchmark we will mimic natural uses of a priority deque. We have chosen two approaches both inspired by actual applications that uses priority queues that might benefit from using priority deques instead:

– Based on the behavior of heap-sort we will start by creating a priority deque and then extract them during the course of the program run. To measure this we will simply add the time needed to create the priority deque and the time needed to extract all elements from it.
– Secondly we will simulate the situation where the elements are added, and extracted from the priority deque continuously during the use of the priority deque. To measure this we have constructed a benchmark routine that adds or extracts an element pseudo-randomly (see the source code in appendix Appendix B). This is e.g. how a kernel process-scheduler behaves.

4.2 Timing Benchmarks

When doing benchmarks it is very important to consider what happens during a run of a benchmark. On a modern multiuser operating system (i.e. UNIX as we use) there will be process-switching, interrupts from I/O devices and other events which will all have an impact on the measured time. UNIX supports the notion of "CPU TIME", i.e. how many CPU-minutes a certain process has used. We would, however, like to achieve a more precise measurement which we can get by measuring the actual running time in microseconds (using the `gettimeofday` function call). In order to minimize the impact of other processes and interrupts we will run the benchmarks on a system with no other users and as little workload as possible.

Since the results might still vary slightly because of interrupts and other events beyond our control, we will repeat all benchmarks five times, remove the fastest and slowest time, and then use the average. This method should eliminate the major uncertainty of the measurements.

4.3 Verification of the results

In order to verify the correctness of our implementation of the algorithms we have implemented a test-program. Our strategy is simply to create a priority deque with each of the algorithms from a given set of numbers. By extracting all the min- or max-elements and comparing with the same element in a sorted array we validate whether the priority deques behaves correctly.

This verification run shows that besides the problem in gcc (see section 3.2), all of our implementations work as they should.

5. Our Expectations of Performance

Before benchmarking we try to speculate on how the three algorithms will perform. In section 6 we will verify whether the tests conform to our predictions.

The worst-case running time of all the algorithms are the same (as described in section 1.1). The deap claims to have a better theoretical time complexity than MinMax-heaps and we would therefore expect it to perform better. Interval-heaps are closely related to standard heaps and we would therefore expect this algorithm to behave approximately similar to standard heaps.

Since there is a very large penalty on cache-misses, the algorithms that utilizes the cache well will get a large advantage. It is interesting to analyze how the different algorithms behaves in this context. Generally for all algorithms we will only look at the overall behavior — not at the details in the individual methods.

The overall cache-use of the algorithms is:

MinMax-heaps compares nodes on tree different levels in the heap simultaneously: It makes comparisons between an element and the child and the grandchild of the node. This requires the value of three nodes for each computation step, which for all but the smallest levels will require three cache-lines[6]. When the algorithm proceeds to the next node that needs computing, it will move two levels up, and therefore have to use two new cache-lines which possibly will result in cache-misses.

[6] The first few levels are small enough to fit in one cache-line.

The deap will for each node work with the parents or children of a node. When the algorithm move up or down the three it needs access to one new level in the tree (that is one cache-line). When reaching the bottom of the data structure and working with a leaf node the algorithm has to compare with the corresponding elements in the other half of the heap. In these cases the total number of used cache-lines is four, and three of them will most likely result in cache-misses.

Interval-heaps keeps the two elements of the interval in neighboring elements, so these two elements will very likely be placed in the same cache-line. If this is the case the interval algorithm will only need two different cache lines at the same time, since it uses either the parent or the children of a node. When moving one step the algorithm will need one new cache-line.

The standard heap does not need to compare to anything else than the parent or child of a node, so it will only work on two cache-lines at the same time, and need one new for each step.

If we look at theoretical running times and the cache-use we will expect the Min-Max to be very slow, the deap to be somewhat faster while interval and standard heap should be fastest.

6. Results of the Benchmarks

This section will give the results of the benchmarks described in section 4 and we will try to compare the results with the expectations we described in section 5. The results of the benchmarks are shown in figure 7 to 12.

6.1 Summary of results

The results of the benchmarks turns out to be very similar in all cases, as the ranking of the algorithms is the same for all benchmarks. We will therefore discuss the results as a whole and not go into details with the individual benchmarks.

Figure 7 is worth some comments, though. A closer examination of the steep part of the curve on the MinMax-graph (we ran some additional measurements) reveals it to be located at an input-size of about 64k integers which is exactly the size of the second-level cache[7]. Thus we interpret the climb as being the point of the input-size exceeding the second-level cache size and thereby increasing the amount of cache-misses substantially. The smaller climb located at 32kB we interpret as a point from where the effectiveness of the cache decreases because of some impact due to the cache-policy — the level 2 cache is 4-way set-associative with an LRU policy. We have not pursued this problem in depth, as it is not the main focus of this report. It does, however, tell something about the MinMax-heap algorithm and that its cache-use impacts performance.

[7] The benchmark-machine has 256kB level 2 cache and one integer has a size of 4 bytes

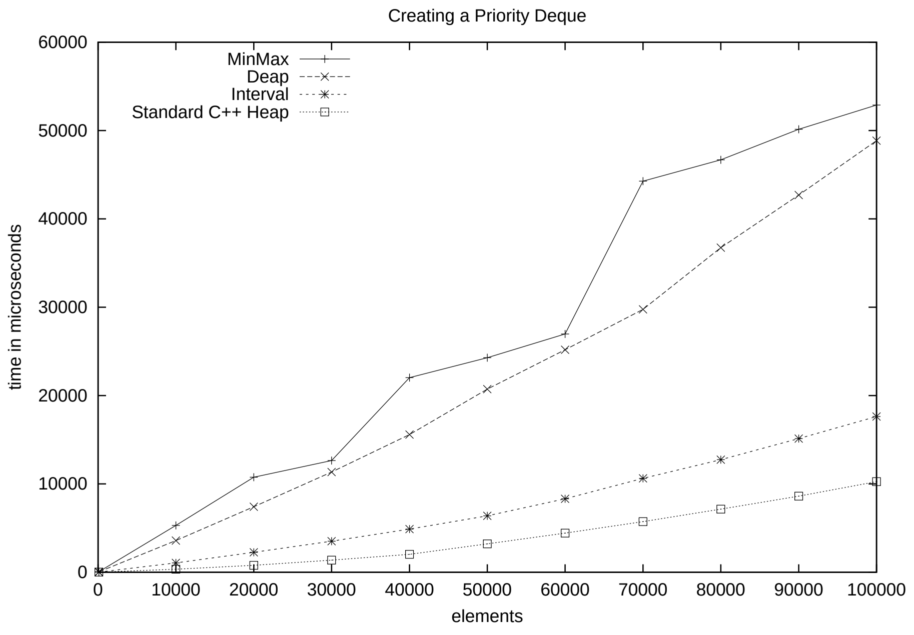

Figure 7. Benchmark of creation of a priority deque.

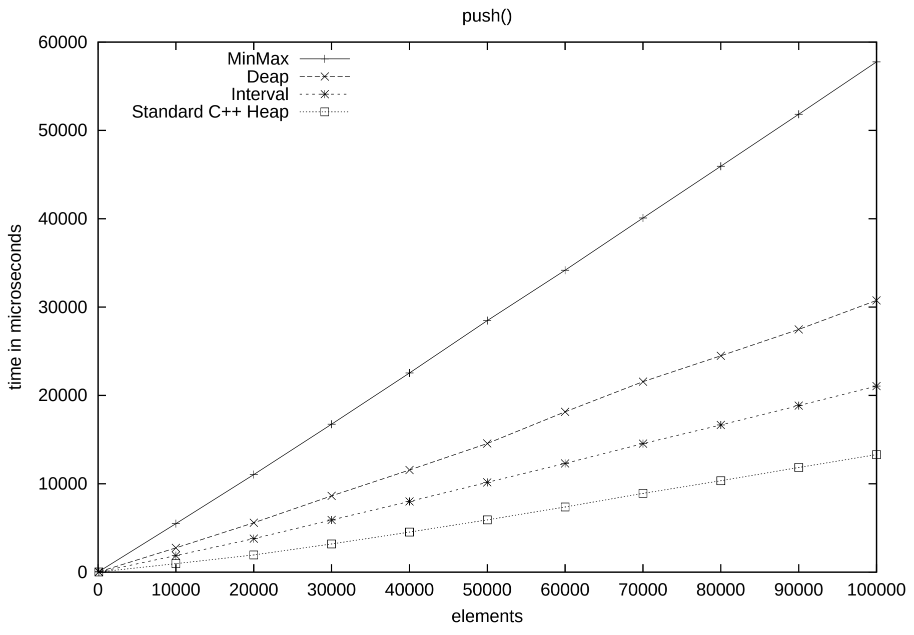

Figure 8. Benchmark of inserting elements in a priority deque.

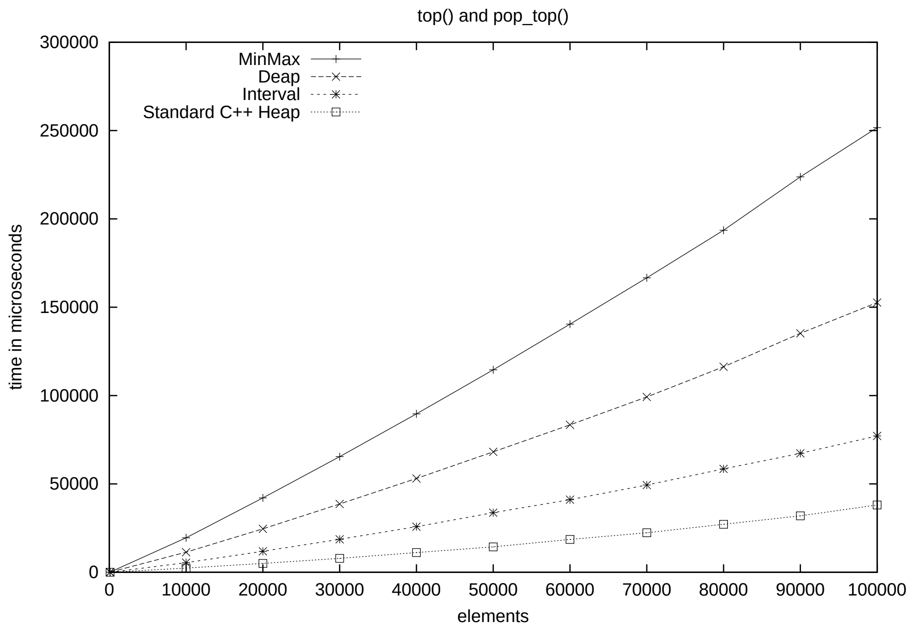

Figure 9. Benchmark of extracting the smallest element from a priority deque.

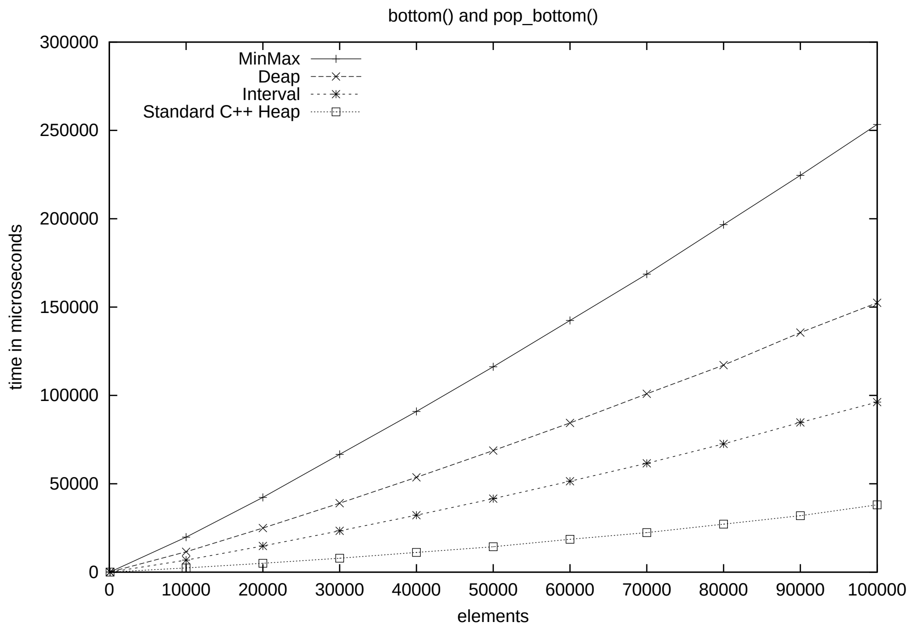

Figure 10. Benchmark of extracting the largest element from a priority deque.

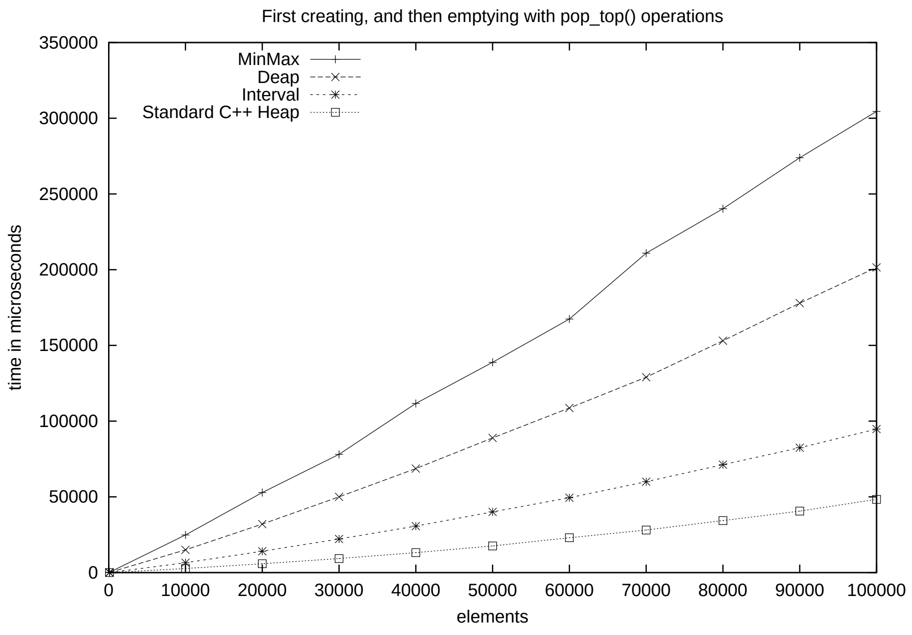

Figure 11. Benchmark of creating and then extracting all elements.

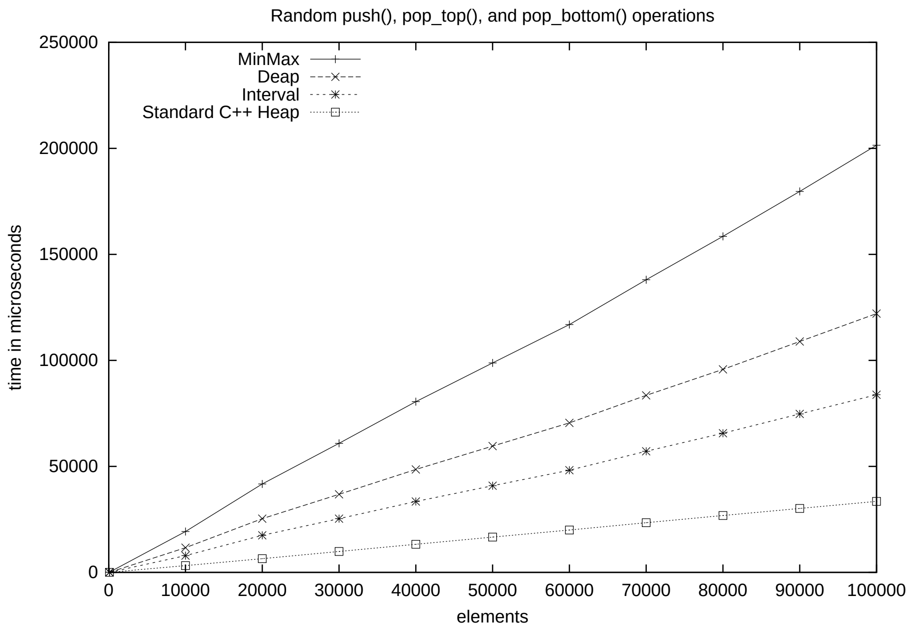

Figure 12. Benchmark of random insert and extract commands

The results of the benchmarks shows a close correlation between our expectations and the results. The MinMax-heap is the slowest closely followed by the deap. These two algorithms are by far slower than interval heaps and the standard heap is the fastest. As described in section 4 standard heaps cannot be compared directly with interval heaps and in this perspective it is not bad to have the interval heap being about twice as slow as the standard heap.

The results show that in all the different uses of priority deques the interval heap is the fastest[8]. If we compare the interval heap with the standard heap we see that the interval heap does add a fairly large overhead, so it is only when needing the extra functionality of a priority deque that the interval heap should be used.

Based on the above result we have made interval the default algorithm in our implementation of our Copenhagen STL priority deque-class (see appendix Appendix B).

7. Conclusion

We have successfully implemented three different priority deques and compared their actual running times in order to choose the best for the Copenhagen STL project.

During the work of understanding and programming the algorithms we discovered shortcomings of Svante Carlsson's "Deap"-algorithm which we then provide solutions for. In seems strange that Carlsson should have overlooked this problem, but we have not been able to find a reasonable explanation of the revealed problems in any of his articles which we have come across.

Our benchmarks show that interval-heaps outperform both MinMax-heaps and the Deap and so we have chosen this as the default algorithm.

7.1 Suggestions for Further Work

Finally, here are some suggestions for further work on this subject:

– We have come up with a suggestion for what we believe to be an new possibility for an algorithm: The basic idea is to hold the elements in a balanced binary search tree. This provides the functionality of creating, inserting and extracting in the required time complexity of priority deques (see 1.1). To be able to read the min- and max-element in constant time, we would maintain pointers to the min- and max elements (Respectively the leftmost and rightmost leaf nodes). This would require extra space for two pointers, which should not be a major issue.

[8] And since all the basic methods on priority deques is fastest in interval heaps, it is fair to assume that all uses, no matter what, are fastest with interval heaps.

An interesting study would be to investigate how the algorithm performs compared with the interval heap. Since binary search trees are based on another fundamental structure, we expect that a binary search tree-based priority deque will have advantages in some of the methods, and possibly have disadvantages in other.

- Our experience is that the use of the cache has a large influence on the performance of an algorithm. A natural improvement of the performance would therefore be to optimize the use of the cache.
  Looking at interval heaps (which is the fastest algorithm and therefore the obvious choice of further refinement) it should be possible to either store more values in the "interval" of a node. For example one node could be made to occupy exactly one cache-line. This would reduce the depth of the tree and thereby minimizing the number of nodes to visit when moving up and down the tree as well as utilizing each memory lookup in an optimal fashion as the entire contents of one interval would be retrieved on one cache-miss.
  Another possibility would be to make a node have several children and it might even be an advantage to combine the two concepts.

References

[1] M. D. Atkinson, J. R. Sack, N. Santoro, and T. Strothotte, Min-max heaps and generalized priority queues, *Communications of the ACM* **29** (1986), 996–1000.

[2] S. Carlsson, Deap — A double-dnded heap to implement double-ended priority queues, Technical report, Lund University (1986).

[3] S. Carlsson, Heaps, Ph. D. Thesis, Department of Computer Science, Lund University, Lund, Sweden (1986).

[4] S. Carlsson, The deap—A double-ended heap to implement double-ended priority queues, *Inf. Process. Lett.* **26** (1987), 33–36.

[5] Department of Computing, University of Copenhagen, The Copenhagen STL, Worldwide Web Document (2001). Available at `http://www.cphstl.dk/`.

[6] International Organization for Standardization (ISO), *ISO/IEC 14882: Standard for the C++ Programming Language*, Genevé (1998).

[7] Silicon Graphics Computer Systems, Inc., Priority_queue<t, sequence, compare>, Worldwide Web Document (2001). Available at `http://www.sgi.com/tech/stl/priority_queue.html`.

[8] Silicon Graphics Computer Systems, Inc., Standard template library programmer's guide, Worldwide Web Document (2001). Available at `http://www.sgi.com/Technology/STL/`.

[9] J. van Leeuwen and D. Wood, Interval heaps, *The Computer Journal* **36**,3 (1993), 209–216.

Appendix A. Work Description

The purpose of the project is to implement the three methods for min/max heaps. Based on these implementations, we will compare the actual running times of the different algorithms.

In contrast to a normal heap a min/max heap is capable of finding both the smallest and the largest element in the heap.

This project will compare the three different suggestions for min/max heaps:

– M. D. Atkinson and J.-R. Sack and N. Santoro and T. Strothotte: "Min-max heaps and generalized priority queues", Communications of the ACM (1986)
– Svante Carlsson: "The deap — A double-ended heap to implement double-ended priority queues", Information Processing Letters, (1987)
– J. van Leeuwen and D. Wood: "Interval Heaps", The Computer Journal, (1993)

We will do this by implementing the three algorithms in C++ in a way which conforms to the STL definitions, so that the best result can be used in Copenhagen STL[9].

These methods describe algorithms for min/max heaps, that all have the same asymptotic time complexity, but the question is how these perform in a "real-world" scenario.

In the comparison of the algorithms, we will try to find out what causes the differences, between the methods. We will e.g. look at the effects of cache-size and type, to see if there is a significant difference between the different algorithms use of cache.

We will also compare the min/max heap algorithms with STL implementations of a normal, heap to determine the extra cost of using min/max heaps instead of normal heaps.

[9] see www.cphstl.dk

Appendix B. Source Code

The source code is included on the following pages. The files are found on page:

File	Page	Description
prioritydequeue.h	26	The file that provides the interface visible to the user.
prioritydequeueinterface.h	27	The common interface for the algorithms.
minmax.h	29	The algorithm MinMax.
deap.h	32	The algorithm deap.
interval.h	35	The algorithm interval.
log2.h	38	Provides a constant time $\log_2$ in g++ and a fast non-constant alternative in other compilers.
test.cpp	39	Our test and benchmark program.

```
#include "minmax.h"
#include "deap.h"
#include "interval.h"


#include<vector>
#include<functional>

#ifndef __PRIORITYDEQUEUE__
#define __PRIORITYDEQUEUE__


using namespace std;

namespace PriorityDequeueNS
{
  template<class T, class Sequence = std::vector<T>,
           class Compare = std::less<T>,
           class PDAlgo = Interval<T, Sequence, Compare> >
    class PriorityDequeue {
      public:

      typedef typename Sequence::size_type size_type;

      typedef T value_type;

      enum child_def {Left = 0, Right = 1};

      bool empty() const {return algo_.empty();}
      size_type size() const{return algo_.size();}

      PriorityDequeue();
      PriorityDequeue(const Compare& comp);
      PriorityDequeue(const T* first, const T* last);
      PriorityDequeue(const T* first, const T* last, const Compare& comp);

      ~PriorityDequeue() {};

      // functions that is dispatched to the actual implementation.
      const T& top() const {return algo_.top();}
      const T& bottom() const {return algo_.bottom();}

      void push(const T& element) {algo_.push(element);}

      void pop_top() {algo_.pop_top();}
      void pop_bottom() {algo_.pop_bottom();};

      private:
      PDAlgo algo_;
  };


  template<class T, class S, class C, class A>
  PriorityDequeue<T,S,C,A>::PriorityDequeue()
    : algo_()
    {}

  template<class T, class S, class C, class A>
  PriorityDequeue<T,S,C,A>::PriorityDequeue(const C& comp)
    : algo_(comp)
    {}
```

```
  template<class T, class S, class C, class A>
  PriorityDequeue<T,S,C,A>::PriorityDequeue(const T* first, const T* last)
    : algo_(first, last)
    {}

  template<class T, class S, class Compare, class A>
  PriorityDequeue<T,S,Compare,A>::
    PriorityDequeue (const T* first, const T* last, const Compare& comp)
    : algo_(first, last, comp)
    {}

} // end namespace
#endif
```

Oct 01, 01 0:14 **prioritydequeueinterface.h** Page 1/3

```
#include<vector>
#include<functional>

#ifndef __PRIORITYDEQUEUEINTERFACE__
#define __PRIORITYDEQUEUEINTERFACE__


using namespace std;

namespace PriorityDequeueNS
{
  template<class T,
    class Sequence = std::vector<T>,
    class Compare = std::less<T> >
  class PriorityDequeueInterface
  {
    public:

    typedef typename Sequence::size_type size_type;
    typedef T value_type;

    enum child_def {Left = 0, Right = 1};

    PriorityDequeueInterface();
    PriorityDequeueInterface(const Compare& comp);
    PriorityDequeueInterface(const T* first,
                             const T* last);
    PriorityDequeueInterface(const T* first,
                             const T* last,
                             const Compare& comp);

    virtual ~PriorityDequeueInterface() {};

    bool empty() const;
    size_type size() const;

  protected:
    size_type size_;
    size_type offset_;
    Sequence heap_;
    Compare comp_;

    // Helpfunctions used by all 3 methods
    virtual size_type child(size_type pos, child_def child);
    inline bool hasChildren(size_type pos);
    inline size_type parent(size_type pos);
    inline size_type lessTh(size_type x, size_type y) const;
    inline size_type moreTh(size_type x, size_type y) const;
    inline void heapswap(size_type x, size_type y);
    virtual void buildHeap(const T* first,
                           const T* last) {}
  };

  template<class T, class S, class C>
  inline PriorityDequeueInterface<T,S,C>::size_type
  PriorityDequeueInterface<T,S,C>::child(size_type pos,
                                        child_def child)
  {
    size_type res;
    if (pos == -1) {return -1;}
    res = pos*2+1+child;
    // make sure that the posistion is smaler than the number of elements.
```

Oct 01, 01 0:14 **prioritydequeueinterface.h** Page 2/3

```
    if (res >= size_ + offset_) {
      return -1;
    }
    return res;
  }

  // Calculates wether the posistion pos has one or more children.
  template<class T, class S, class C>
  inline bool PriorityDequeueInterface<T,S,C>::hasChildren(size_type pos)
  {
    return (child(pos, Left) != -1);
  }

  template<class T, class S, class C>
  inline PriorityDequeueInterface<T,S,C>::size_type
  PriorityDequeueInterface<T,S,C>::parent(size_type pos)
  {
    if (pos == -1 || pos <= offset_*2) return - 1;
    return (pos+1)/2 - 1;
  }

  template<class T, class S, class C>
  inline PriorityDequeueInterface<T,S,C>::size_type
  PriorityDequeueInterface<T,S,C>::lessTh(size_type x, size_type y) const
  {
    if (y == -1 || y >= size_ + offset_) return x;
    return comp_(heap_[x], heap_[y]) ? x : y;
  }

  template<class T, class S, class Compare>
  inline PriorityDequeueInterface<T,S,Compare>::size_type
  PriorityDequeueInterface<T,S,Compare>::moreTh(size_type x,
                                               size_type y) const
  {
    if (y == -1 || y >= size_ + offset_) return x;
    return comp_(heap_[x], heap_[y]) ? y : x;
  }

  template<class T, class S, class C>
  inline void PriorityDequeueInterface<T,S,C>::heapswap(size_type x,
                                                        size_type y)
  {
    size_type tmp = heap_[x];
    heap_[x] = heap_[y];
    heap_[y] = tmp;
  }

  template<class T, class S, class C>
  inline bool PriorityDequeueInterface<T,S,C>::empty() const
  {
    return (!size_);
  }

  template<class T, class S, class C>
  inline PriorityDequeueInterface<T,S,C>::size_type
  PriorityDequeueInterface<T,S,C>::size() const
  {
    return size_;
  }

  template<class T, class S, class C>
  PriorityDequeueInterface<T,S,C>::PriorityDequeueInterface()
```

```
    : size_(0), heap_() {}

  template<class T, class S, class C>
  PriorityDequeueInterface<T,S,C>::PriorityDequeueInterface
  (const C& comp)
    : size_(0), heap_(), comp_(comp) {}

  template<class T, class S, class C>
  PriorityDequeueInterface<T,S,C>::PriorityDequeueInterface
  (const T* first, const T* last)
    : size_(0), heap_(first, last)
  {}

  template<class T, class S, class Compare>
  PriorityDequeueInterface<T,S,Compare>::PriorityDequeueInterface
  (const T* first, const T* last,
   const Compare& comp)
    : size_(0), heap_(first, last), comp_(comp)
  {}

} // end namespace



#endif
```

```
#ifndef __MINMAXDEQUEUE__
#define __MINMAXDEQUEUE__

#include"prioritydequeueinterface.h"
#include"log2.h"

#include<vector>
#include<functional>
#include<stdio.h>


namespace PriorityDequeueNS
{
  template<class T, class Sequence, class Compare>
    class MinMax : public PriorityDequeueInterface<T, Sequence, Compare>
    {
      private:
      bool onMinLevel(size_type pos);
      void trickleDownMin(size_type pos);
      void trickleDownMax(size_type pos);
      void trickleDown(size_type pos);
      void bubbleUpMin(size_type pos);
      void bubbleUpMax(size_type pos);
      void bubbleUp(size_type pos);

      void buildHeap(const T* first, const T* last);


      public:
      MinMax() : PriorityDequeueInterface<T,Sequence,Compare>()
      {
        offset_ = 0; // Indicate that the container are indexed from 0.
      };

      MinMax(const Compare& comp)
      : PriorityDequeueInterface<T,Sequence,Compare>(comp)
      {
        offset_ = 0; // Indicate that the container are indexed from 0.
      };

      MinMax(const T* first, const T* last)
      : PriorityDequeueInterface<T,Sequence,Compare>(first, last)
      {
        offset_ = 0; // Indicate that the container are indexed from 0.
        buildHeap(first, last);
      };

      MinMax(const T* first, const T* last,
             const Compare& comp)
      : PriorityDequeueInterface<T,Sequence,Compare>(first, last, comp)
      {
        offset_ = 0; // Indicate that the container are indexed from 0.
        buildHeap(first, last);
      };

      // The functions offered as the interface.
      const T& top() const;
      const T& bottom() const;

      void push(const T& element);
```

```
      void pop_top();
      void pop_bottom();
    };

  // Calculates whether the position pos is on a Min Level.
  template<class T, class S, class C>
  inline bool MinMax<T,S,C>::onMinLevel(size_type pos)
  {
    size_type res = // floor_log2<size_type>(pos+1);
                    (size_type)floor(log(pos+1)/log(2));

    return !(res & 1);  // The last bit is 0
  }

  template<class T, class S, class C>
  void MinMax<T,S,C>::trickleDownMin(size_type pos)
  {
    if (hasChildren(pos))
      {
        // Find the smalest of children and grandchildren.
        size_type m;
        size_type cl = child(pos, Left);
        size_type cr = child(pos, Right);


        m = lessTh(cl, cr);
        m = lessTh(m, child(cl,Left));
        m = lessTh(m, child(cl,Right));
        m = lessTh(m, child(cr,Left));
        m = lessTh(m, child(cr,Right));

        // Is m a grandchild?
        if (m > cr)
          {
            if (comp_(heap_[m], heap_[pos]))
              {
                heapswap(m, pos);
                if (comp_(heap_[parent(m)], heap_[m]))
                  {
                    heapswap(m, parent(m));
                  }
                trickleDownMin(m);
              }
          }
        else // when m is a child of pos
          {
            if (comp_(heap_[m], heap_[pos]))
              {
                heapswap(m, pos);
              }
          }
      }
  }

  template<class T, class S, class C>
  void MinMax<T,S,C>::trickleDownMax(size_type pos)
  {
    if (hasChildren(pos))
      {
        // Find the largest of children and grandchildren.
        size_type m;
        size_type cl = child(pos, Left);
      size_type cr = child(pos, Right);

      m = moreTh(cl, cr);
      m = moreTh(m, child(cl,Left));
      m = moreTh(m, child(cl,Right));
      m = moreTh(m, child(cr,Left));
      m = moreTh(m, child(cr,Right));

      // Is m a grandchild?
      T heapM = heap_[m];
      if (m > cr)
        {
          if (comp_(heap_[pos], heapM))
            {
              heapswap(m, pos);
              if (comp_(heap_[m], heap_[parent(m)]))
                {
                  heapswap(m, parent(m));
                }
              trickleDownMax(m);
            }
        }
      else // when m is a child of pos
        {
          if (comp_(heap_[pos], heapM))
            {
              heapswap(m, pos);
            }
        }
    }
}

template<class T, class S, class C>
void MinMax<T,S,C>::trickleDown(size_type pos)
{
  if (onMinLevel(pos))
    {
      trickleDownMin(pos);
    }
  else
    {
      trickleDownMax(pos);
    }
}

template<class T, class S, class C>
void MinMax<T,S,C>::bubbleUpMin(size_type pos)
{
    if (pos > 2) // Pos has a grandparant.
    {
      size_type grand = parent(parent(pos));
      if (comp_(heap_[pos], heap_[grand]))
        {
          heapswap (pos, grand);
          bubbleUpMin(grand);
        }
    }
}

template<class T, class S, class C>
void MinMax<T,S,C>::bubbleUpMax(size_type pos)
{
```

```
  if (pos > 3) // Pos has a grandparant.
    {
      size_type grand = parent(parent(pos));
      if (comp_(heap_[grand],heap_[pos]))
        {
          heapswap (pos, grand);
          bubbleUpMax(grand);
        }
    }
}

template<class T, class S, class C>
void MinMax<T,S,C>::bubbleUp(size_type pos)
{
  size_type pare = parent(pos);

  if (onMinLevel(pos))
    {
      // If pos has a parant, and the parants value is larger.
      if (pos != 0 && comp_(heap_[pare],heap_[pos]))
        {
          heapswap(pos, pare);
          bubbleUpMax(pare);
        }
      else
        {
          bubbleUpMin(pos);
        }
    }
  else // on a Max level
    {
      // If pos has a parant, and the parants value is larger.
      if ((pos != 0) && comp_(heap_[pos], heap_[pare]))
        {
          heapswap(pos, pare);
          bubbleUpMin(pare);
        }
      else
        {
          bubbleUpMax(pos);
        }
    }
}

template<class T, class S, class C>
inline const T& MinMax<T,S,C>::top() const
{
  return heap_[0];
}

template<class T, class S, class C>
inline const T& MinMax<T,S,C>::bottom() const
{
  if (size_ == 1) {return heap_[0];}
  return heap_[moreTh(1,2)];
}

template<class T, class S, class C> void MinMax<T,S,C>::pop_top()
{
  heap_[0] = heap_[size_-1];
  size_--;
  heap_.erase(heap_.end()-1);
```

```
    trickleDown(0);
  }

  template<class T, class S, class C> void MinMax<T,S,C>::pop_bottom()
  {
    if (size_ <= 1) {pop_top();return;}
    size_type pos = moreTh(1,2);
    heap_[pos] = heap_[size_−1];
    size_−−;
    heap_.erase(heap_.end()−1);
    trickleDown(pos);
  }

  template<class T, class S, class C>
  void MinMax<T,S,C>::buildHeap(const T* first, const T* last)
  {
    size_ = (last − first);
    // find the first nontrivial level to trickleDown.
    size_type j = (1 << (size_type)floor(log(size_)/log(2))) − 2;
    for (; j != −1; j−−)
      {
        trickleDown(j);
      }
  }

  template<class T, class S, class C>
  void MinMax<T,S,C>::push(const T& elem)
  {
    // insert element in first free slot.
    heap_.insert(heap_.end(), elem);
    size_++;
    // Reestablish min−max ordering.
    bubbleUp(size_−1);
  }

} // namespace
#endif
```

```
#ifndef __DEAP__
#define __DEAP__

#include"prioritydequeueinterface.h"
#include"log2.h"

#include<vector>
#include<functional>
#include<stdio.h>


namespace PriorityDequeueNS
{
  template<class T, class Sequence, class Compare>
    class Deap : public PriorityDequeueInterface<T, Sequence, Compare>
    {
      private:
      bool inMinHeap(size_type pos);
      inline size_type halfWidth(size_type pos);

      void pullUpMin(size_type i, const T X);
      void pullUpMax(size_type i, const T X);

      void trickleUpMin(size_type index, size_type topindex, const T X);
      void trickleUpMax(size_type index, size_type topindex, const T X);

      void buildHeap(const T* first, const T* last);

      void insertMin(size_type i, size_type topindex, const T X);
      void insertMax(size_type i, size_type topindex, const T X);

      public:
      Deap() : PriorityDequeueInterface<T,Sequence,Compare>()
      {
        offset_ = 1; // Indicate that the container is indexed from 1.

        // Insert empty element in heap_[0], which is never used.
        heap_.insert(heap_.end(), T());
      };

      Deap(const Compare& comp)
      : PriorityDequeueInterface<T,Sequence,Compare>(comp)
      {
        offset_ = 1; // Indicate that the container is indexed from 1.

        // Insert empty element in heap_[0], which is never used.
        heap_.insert(heap_.end(), T());
      };

      Deap(const T* first, const T* last)
      : PriorityDequeueInterface<T,Sequence,Compare>(first, last)
      {
        offset_ = 1; // Indicate that the container is indexed from 1.

        // The elements are initialised in [0,size[ move it to [1,size+1[
        heap_.insert(heap_.end(), heap_[0]);
        heap_[0]=-1;

        buildHeap(first, last);
      };

      Deap(const T* first, const T* last, const Compare& comp)
```

```
      : PriorityDequeueInterface<T,Sequence,Compare>(first, last, comp)
      {
        offset_ = 1; // Indicate that the container is indexed from 1.

        // The elements are initialised in [0,size[ move it to [1,size+1[
        heap_.insert(heap_.end(), heap_[0]);
        heap_[0]=-1;

        buildHeap(first, last);
      };

      // virtual functions for the individual methods.
      const T& top() const;
      const T& bottom() const;

      void push(const T& element);

      void pop_top();
      void pop_bottom();
    };

  // Calculates whether the position pos is on a Min Level.
  template<class T, class S, class C>
  inline bool Deap<T,S,C>::inMinHeap(size_type pos)
  {
    size_type level = floor_log2<size_type>(pos+1);

    if (level == 1)
      {
        return (pos==1);
      }
    size_type heapMiddle = (3 << (level-1))-1;
    return (pos < heapMiddle);
  }

  template<class T, class S, class C>
  inline Deap<T,S,C>::size_type Deap<T,S,C>::halfWidth(size_type pos)
  {
    size_type level = floor_log2<size_type>(pos+1);

    if (level == 1)
      {
        return (pos==2) ? 1 : 2;
      }
    return 2 << (level-2);
  }

  template<class T, class S, class C>
  void Deap<T,S,C>::push(const T& X)
  {
    if (size_ == 0)
      {
        heap_.insert(heap_.end(), X);
      }
    if (size_ == 1)
      {
        if (comp_(X, heap_[1]))
          {
            heap_.insert(heap_.end(), heap_[1]);
            heap_[1] = X;
          }
        else    // X is larger.
        {
          heap_.insert(heap_.end(), X);
        }
    }
  if (size_ > 1)
    {
      heap_.insert(heap_.end(), X);
      if (inMinHeap(size_ + 1))
        {
          insertMin(size_ + 1, 1, X);
        }
      else
        {
          insertMax(size_ + 1, 1, X);
        }
    }
  size_++;
}




template<class T, class S, class C>
void Deap<T,S,C>::trickleUpMin(size_type index, size_type topindex,
                               const T X)
{
  size_type papa = parent(index);

  while (papa != −1 && papa >= topindex &&
         comp_(X, heap_[papa]))
    {
      heap_[index] = heap_[papa];
      index = papa;
      papa = parent(index);
    }
  heap_[index] = X;
}


template<class T, class S, class C>
void Deap<T,S,C>::trickleUpMax(size_type index, size_type topindex,
                               const T X)
{
  size_type papa = parent(index);

  while (papa != −1 && papa >= topindex &&
         comp_(heap_[papa], X))
    {
      heap_[index] = heap_[papa];
      index = papa;
      papa = parent(index);
    }
  heap_[index] = X;
}

template<class T, class S, class C>
void Deap<T,S,C>::insertMin(size_type index, size_type topindex, const T X)
{
  size_type cores = (index + halfWidth(index));
  if (cores > size_)
    {
      cores = (cores − 1) >> 1; // coresponding / 2
```

```
    }
  if (child(index,Left) == −1) // Index is a leaf node. Check coresponding.
    {
      if (comp_(heap_[cores], X))
        {  // The coresponding is larger => swap
          heap_[index] = heap_[cores];
          trickleUpMax(cores, topindex, X);
          return;
        }
    }
  // If index not a leaf node, or if coresponding is smaller.
  trickleUpMin(index, topindex, X);
}


template<class T, class S, class C>
void Deap<T,S,C>::insertMax(size_type index, size_type topindex, const T X)
{
  size_type cores = index − halfWidth(index);
  if (child(index,Left) == −1) // Index is a leaf node. Check coresponding.
    {

      // DO WE HAVE SPECIAL CASE 2 ?
      if (index == size_ && index%2 == 1 && parent(index) >= topindex)
        // index does not have a sister, and index is a left child.
        {
          if (comp_(heap_[parent(index)], heap_[cores+1]))
            {
              T tmp = heap_[parent(index)];
              heap_[parent(index)] = heap_[cores+1];
              heap_[cores+1] = tmp;
            }
        }



      // SPECIAL CASE: Handle that up to 3 elements are coresponding.
      if (child(cores,Left) != −1)
        {  // The swap element has children, so compare with the largest.
          cores  = moreTh(child(cores,Left),child(cores,Right));
        }

      if (comp_(X, heap_[cores]))
        {  // The coresponding is larger => swap
          heap_[index] = heap_[cores];
          trickleUpMin(cores, topindex, X);
          return;
        }
    }
  // Index is not a leaf node, or coresponding is smaller that X.
  trickleUpMax(index, topindex, X);
}

template<class T, class S, class C>
inline const T& Deap<T,S,C>::top() const
{
  return heap_[1];
}

template<class T, class S, class C>
inline const T& Deap<T,S,C>::bottom() const
{
    if (size_ == 1) return heap_[1];
    return heap_[2];
  }

  template<class T, class S, class C>
  void Deap<T,S,C>::pullUpMin(size_type i, const T X)
  {
    size_type empty = i;
    while(child(empty,Left) != -1)
      {
        size_type min = lessTh(child(empty,Left), child(empty,Right));
        heap_[empty] = heap_[min];
        empty = min;
      }
    insertMin(empty,i,X);
  }

  template<class T, class S, class C>
  void Deap<T,S,C>::pullUpMax(size_type i, const T X)
  {
    size_type empty = i;
    while(child(empty,Left) != -1)
      {
        size_type max = moreTh(child(empty,Left), child(empty,Right));
        heap_[empty] = heap_[max];
        empty = max;
      }
    insertMax(empty,i,X);
  }

  template<class T, class S, class C>
  void Deap<T,S,C>::pop_top()
  {
    pullUpMin(1, heap_[size_]);
    heap_.erase(heap_.end()-1);
    size_--;
  }


  template<class T, class S, class C>
  void Deap<T,S,C>::pop_bottom()
  {
    size_--;
    pullUpMax(2, heap_[size_+1]);
    heap_.erase(heap_.end()-1);
  }



  template<class T, class S, class C>
  void Deap<T,S,C>::buildHeap(const T* first, const T* last)
  {
    size_ = (last - first);
    for (size_type j = size_; j != 0; j--)
      {
        if (inMinHeap(j))
          {
            pullUpMin(j, heap_[j]);
          }
        else
          {
            pullUpMax(j, heap_[j]);
```

```
          }
      }
  }

} // namespace

#endif
```

```
#ifndef __INTERVAL__
#define __INTERVAL__

#include"prioritydequeueinterface.h"

#include<vector>
#include<functional>
#include<stdio.h>


namespace PriorityDequeueNS
{
  template<class T, class Seq, class Compare>
    class Interval : public PriorityDequeueInterface<T, Seq, Compare>
  {
    private:
    inline size_type begin(size_type index);
    inline size_type end(size_type index);

    inline size_type minChild(size_type index, child_def child);
    inline size_type maxChild(size_type index, child_def child);

    inline void intervalSwap(size_type index);

    void heapifyMin(size_type index);
    void heapifyMax(size_type index);

    void boubleUpMin(size_type index);
    void boubleUpMax(size_type index);

    void buildHeap(const T* first, const T* last);

    public:
    Interval() : PriorityDequeueInterface<T,Seq,Compare>()
    {
      offset_ = 0; // Indicate that the container is indexed from 0.
    };

    Interval(const Compare& comp)
    : PriorityDequeueInterface<T,Seq,Compare>(comp)
    {
      offset_ = 0; // Indicate that the container is indexed from 0.
    };

    Interval(const T* first, const T* last)
    : PriorityDequeueInterface<T,Seq,Compare>(first, last)
    {
      offset_ = 0; // Indicate that the container is indexed from 1.

      buildHeap(first, last);
    };

    Interval(const T* first, const T* last,
             const Compare& comp)
    : PriorityDequeueInterface<T,Seq,Compare>(first, last, comp)
    {
      offset_ = 0; // Indicate that the container is indexed from 1.
      buildHeap(first, last);
    };

    ~Interval() {};
```

```
    // virtual functions for the individual methods.
    const T& top() const;
    const T& bottom() const;

    void push(const T& element);

    void pop_top();
    void pop_bottom();
  };

  template<class T, class S, class C>
    inline Interval<T,S,C>::size_type Interval<T,S,C>::begin(size_type index)
    {
      return (index << 1);
    }

  template<class T, class S, class C>
    inline Interval<T,S,C>::size_type Interval<T,S,C>::end(size_type index)
    {
      size_type res = index << 1;
      if (res != size_−1)
        {
          res++;
        }
      return res;
    }

  template<class T, class S, class C>
    inline Interval<T,S,C>::size_type
    Interval<T,S,C>::minChild(size_type index, child_def child)
    {
      size_type res;
      if(index == −1) {return −1;}

      res = ((index+1) << 1)+(child << 1);

      // make sure that the posistion is smaler than the number of elements.
      if (res >= size_) {
        return −1;
      }
      return res;
    }

  template<class T, class S, class C>
    inline Interval<T,S,C>::size_type
      Interval<T,S,C>::maxChild(size_type index, child_def child)
    {
      size_type res;
      if(index == −1) {return −1;}

      res = ((index)*2)+(child*2)+1;
      // make sure that the posistion is smaler than the number of elements.
      if (res == size_) { return res − 1; } // Use the min element as psudomax e
lement.
      if (res > size_) { return −1;}
      return res;
    }

  template<class T, class S, class C>
    inline void Interval<T,S,C>::intervalSwap(size_type index)
    {
      if (comp_(heap_[index+1],heap_[index]))
        {
          heapswap(index, index+1);
        }
    }

  template<class T, class S, class C>
    void Interval<T,S,C>::heapifyMin(size_type index)
    {
      while (minChild(index,Left) != -1)
        {
          size_type min = lessTh(minChild(index,Left), minChild(index,Right));
          if (comp_(heap_[min], heap_[index]))
            {
              heapswap(min,index);
              index = min;
              intervalSwap(min);
            }
          else
            {
              break;
            }
        }
    }

  template<class T, class S, class C>
    void Interval<T,S,C>::heapifyMax(size_type index)
    {
      while (maxChild(index,Left) != -1)
        {
          size_type max = moreTh(maxChild(index,Left), maxChild(index,Right));
          if (comp_(heap_[index], heap_[max]))
            {
              heapswap(max,index);
              index = max;
              intervalSwap(max-1);
            }
          else
            {
              break;
            }
        }
    }

  template<class T, class S, class C>
    void Interval<T,S,C>::buildHeap(const T* first, const T* last)
    {
      size_ = (last - first);
      size_type j = (size_ >> 1) -1;
      for (; j != -1; j--)
        {
          intervalSwap(begin(j));

          heapifyMin(begin(j));
          heapifyMax(end(j));
        }
    }

  template<class T, class S, class C>
    void Interval<T,S,C>::push(const T& X)
    {
      size_++;
```

```
      int index = size_-1 >> 1; // Gives index to the node containing X.

      heap_.insert(heap_.end(),X);  // Insert X.
      if (size_ > 1) {
        if (size_ & 1)  // If the last bit set?
          { // The new element is in a L-node.
            if (comp_(heap_[size_-1], heap_[begin(parent(index))]))
              {
                boubleUpMin(index);
              }
            else
              {
                boubleUpMax(index);
              }
          }
        else
          {  // The new node is in a normal node.
            if (comp_(heap_[size_-1],heap_[size_-2]))
              {
                heapswap(size_-1,size_-2);
                boubleUpMin(index);
              }
            else
              {
                boubleUpMax(index);
              }
          }
      }
    }

  template<class T, class S, class C>
    void Interval<T,S,C>::boubleUpMin(size_type index)
    {
      while (parent(index) != -1 &&
             comp_(heap_[begin(index)], heap_[begin(parent(index))]))
        {
          heapswap(begin(index), begin(parent(index)));
          index = parent(index);
        }
    }

  template<class T, class S, class C>
    void Interval<T,S,C>::boubleUpMax(size_type index)
    {
      while (parent(index) != -1 &&
             comp_(heap_[end(parent(index))], heap_[end(index)]))
        {
          heapswap(end(index), end(parent(index)));
          index = parent(index);
        }
    }

  template<class T, class S, class C>
    inline const T& Interval<T,S,C>::top() const
    { return heap_[0];}

  template<class T, class S, class C>
    inline const T& Interval<T,S,C>::bottom() const
    {
      return (size_ == 1) ? heap_[0] : heap_[1];
    }
  template<class T, class S, class C>
    void Interval<T,S,C>::pop_top()
    {
      heap_[0] = heap_[size_ - 1];
      size_--;
      heap_.erase(heap_.end()-1);
      heapifyMin(0);
    }

  template<class T, class S, class C>
    void Interval<T,S,C>::pop_bottom()
    {
      size_--;
      if (size_>0)
        {
          heap_[1] = heap_[size_];
          heapifyMax(1);
        }
      heap_.erase(heap_.end()-1);
    }

} // namespace

#endif
```

```
/* $Id: code.ps,v 1.1 2002/05/02 10:13:13 dunkel Exp $ */
/*
 * floor_log2: function for calculating floor(lg(x))
 *
 */
#ifndef __log2_h__
#define __log2_h__


#ifdef __GNUC__
//if we are using gnu cc, then we have a library function.
//I believe it works on all types
#include <cmath>
template <typename T, int size>
inline T internal_floor_log2(T x)
{
  return ilogb(x);
}

#else
//otherwise use Sofus' binary search-like impl.
//works for 32-bit integers
//NOTE: does NOT specialize as wanted for 32 bits
template <typename T, int size>
T internal_floor_log2(T x) {
  static const unsigned char log_table[256] = {
    0xff, // <--- rogue value
    0,
    1, 1,
    2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7
  };
  long rv = 0;
  if (x & 0xffff0000) { rv += 16; x >>= 16; }
  if (x & 0xff00) { rv += 8; x >>= 8; }
  if (x & 0xf0) { rv += 4; x >>= 4; }
  if (x & 0xc) { rv += 2; x >>= 2; }
  if (x & 0x2) { rv += 1; x >>= 1; }
  return rv + log_table[x];
}
#endif

template <typename T>
inline T floor_log2(T x)
{
  return internal_floor_log2<T, sizeof(T)>(x);
}
```

```


#endif
```

```cpp
#include "prioritydequeue.h"

#include<stdio.h>
#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<string.h>
#include<vector>
#include<algorithm>
#include<sys/time.h>
#include<queue>

using namespace std;
using namespace PriorityDequeueNS;

#define START  gettimeofday(&tv, 0); beginMS = tv.tv_usec; beginS = tv.tv_sec;
#define STOP   gettimeofday(&tv, 0);   endMS = tv.tv_usec;   endS = tv.tv_sec;
#define PRINT  cout <<  (endS − beginS) * 1000000 + (endMS − beginMS) << "\n";

vector<int> readfile(char* file) {
  vector<int> inddata;
  ifstream in;
  in.open(file);
  if (!in) {
    cout << "Input file <" << file << "> cannot be opened.\n";
    return inddata;
  }
  int number;
  int i = 0;
  while (!in.eof()) {
    in >> number;
    inddata.push_back(number);
    i++;
  }
  inddata.pop_back();
  in.close();
  return inddata;
}


int main(int argc, char** argv)
{
  // vars for timemesurement.
  double beginMS;
  double beginS;
  double endMS;
  double endS;
  struct timeval tv;

  if (argc !=3)
    {
      cout << "Usage: " << argv[0] << " <input file> <method> \n";
      exit(1);
    }

  vector<int> inddata = readfile((argv[1]));

#ifdef DEBUG
  vector<int> sorted(inddata);

  sort(sorted.begin(), sorted.end());
#endif  // DEBUG
```

```cpp

  // Create the heap and time it.
  START;

  // Create the heap.
#ifdef MINMAX
  PriorityDequeue<int, std::vector<int>, std::less<int>,
                  MinMax<int, std::vector<int>, std::less<int> > >
    testheap(inddata.begin(), inddata.end());
#endif // MINMAX

#ifdef DEAP
  PriorityDequeue<int, std::vector<int>, std::less<int>,
                  Deap<int, std::vector<int>, std::less<int> > >
    testheap(inddata.begin(), inddata.end());
#endif // DEAP

#ifdef INTERVAL
  PriorityDequeueNS::PriorityDequeue<int> testheap(inddata.begin(),
                                                   inddata.end());
#endif // DEAP

#ifdef STD
  priority_queue<int> testheap(inddata.begin(), inddata.end());
#endif // STD

  STOP;

  if (!strcmp(argv[2], "create"))
    {
      PRINT;
    }

  if (!strcmp(argv[2], "popmin") || !strcmp(argv[2], "push")
      || !strcmp(argv[2], "rand"))
    {
      START;
      int limit = testheap.size();
      for (int i = 0;i<limit;i++)
        {
#ifdef DEBUG
          if (testheap.top() != sorted[i])
            {
              cout << "Error in get_top/pop_top \n";
              printf("i=%d, topmin=%d, sorted=%d\n",i, testheap.top(), sorted[i]);
              exit(3);
            }
#endif //DEBUG
#ifdef STD
          testheap.top();
          testheap.pop();
#else
          testheap.top();
          testheap.pop_top();
#endif
        }
      STOP;
      if (!strcmp(argv[2], "popmin"))   // print result.
        {
          PRINT;
        }
    }

  if (!strcmp(argv[2], "popmax"))
    {
      START;
      int limit = testheap.size();
      for (int i = limit-1; i>=0 ;i--)
        {
#ifdef DEBUG
          if (testheap.bottom() != sorted[i])
            {
              cout << "Error in get_bottom/pop_bottom \n";
              printf("i=%d, topmax=%d, sorted=%d\n",i, testheap.bottom(), sorted[i]);
              exit(3);
            }
#endif //DEBUG
#ifdef STD
          cout << "You cant popmax on STD \n";
#else
          testheap.bottom();
          testheap.pop_bottom();
#endif
        }
      STOP;
      PRINT;
    }

  if (!strcmp(argv[2], "push"))
    {

      START;
      // Push the elements.
      for (int i=0; i < inddata.size(); i++)
        {
          testheap.push(inddata[i]);
        }
      STOP;
      PRINT;

#ifdef DEBUG
      int limit = testheap.size();
      for (int i = 0;i<limit;i++)
        {
          if (testheap.top() != sorted[i])
            {
              cout << "Error in push \n";
              printf("i=%d, topmin=%d, sorted=%d\n",i, testheap.top(),
                     sorted[i]);
              exit(3);
            }
          testheap.pop_top();
        }
#endif
    }

  if (!strcmp(argv[2], "rand"))
    {
      int limit = inddata.size();
      int size = 0;
      START
      int i = 0;
      while(1)
```

```
        {
          int toDo = (rand()%4); // give a random number in the range [0,3]

          if (toDo < 2) // 0 and 1 is push
            {
              if (i>=limit) {break;}
              testheap.push(inddata[i]);
              i++;
              size++;
            }
#ifndef STD
          if (toDo == 2) // popmin
            {
              if (!testheap.empty())
                {
                  testheap.top();
                  testheap.pop_top();
                }
            }
          if (toDo == 3) // popmax
            {
              if (!testheap.empty())
                {
                  testheap.bottom();
                  testheap.pop_bottom();
                }
            }
          size--;
#else
          if (size)
            {
              testheap.top();
              testheap.pop();
              size--;
            }
#endif
        }
    STOP;
    PRINT;
    }
} // end Main
```