Priority queue and heap functions

Brian S. Jensen

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
purple@diku.dk

1. Introduction

The goal of the Copenhagen STL project, is to develop an enhanced edition of the C++ Standard Template Library. More specifically, we aim to surpass the performance of SGI's widely used implementation of STL. We expect that our version of the library will be used by industry and academic developers worldwide.

The aim of this sub-project is to develop an enhanced version of the priority queue class as well as the heap functions. Our implementation of the priority queue will perform its insert, delete and build operations faster than the SGI implementation.

We will try out several different algorithms and implementations to find the fastest one. To ensure reasonably certain test results, we will use several different benchmark tests and run these on several machines with different architectures.

2. Algorithms

In the following sections we will present 4 different priority queue algorithms, including the one used in SGI STL. All of these will be implemented and tested on several different machines, to see which is the fastest one.

First we need to clear up a few matters of terminology. A priority queue is an ordered sequence of elements, where only the front element is accessible from the outside. In many presentations the front element is called the minimum or the maximum, depending on how the elements are ordered. We will call it the minimum element. Sometimes we will also refer to it as the top element. This is due to the method name used in STL, and because a priority queue is often implemented with some sort of tree. For that reason, it is sometimes convenient to think of the front of the queue as the top of a tree.

CPH STL Report 2001-3, February 2001.

2.1 The binary heap

Priority queues are often implemented using an optimal binary leftist tree that satisfies a heap condition. 'Optimal' means that all internal nodes, with the exception of the lowest ones, must all have two children. 'Leftist' means that all nodes on each level are as far to the left as possible. The heap condition says that all nodes must have a key value which is lower than that of each of its children. To keep things simple, we will just call this type of tree a **binary heap**.

To use the binary heap for a priority queue, we need to implement three operations: Insertion of a new node, reading the value of the minimum node, and deletion of the minimum node. Reading the value of the minimum node is easy, because this node will always be the root of the tree. This also gives us a good starting point for deleting the minimum element. To do this, we remove the root node, and replace it with the node which is at the bottom right of the tree. We then compare this node's value with the value of its two children. If the heap condition is not satisfied, we swap the node with the smallest of its children. We continue this process until the node is in such a position that the heap condition is satisfied. These swaps are also know as **bubble down** operations (because tree nodes are usually drawn as circles, so the replacement node looks like a bubble passing down through the tree).

The insert operation works in a similar way. We insert the new node at the bottom right of the tree. We then compare the value of the new node to its parent. If the heap condition is not satisfied, we swap the node with the parent. This process continues until the heap condition is satisfied. These swaps are also known as **bubble up** operations.

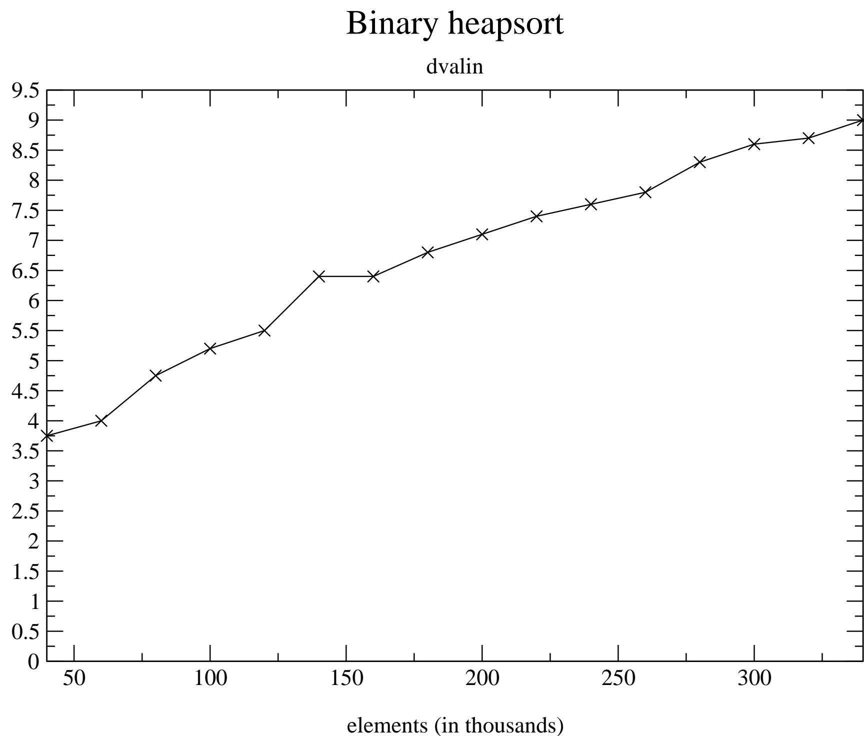
The binary heap is used in SGI's version of STL for implementing the priority queue. Our goal is therefore to find and implement an algorithm which performs better than this binary heap.

2.2 Cache considerations

According to theory, the time used for insertion and minimum element deletion in the traditional binary heap, increases logarithmically in relation to the number of elements in the heap. More precisely these operations take time $O(\log_2 n)$, where n is the number of elements in the heap. This theoretical running time, is not too far from accurate when a cache-less non-superscalar computer is used. However, nowadays most computers have several cache levels as part of their primary memory system. As noted by LaMarca and Ladner in [3], the existence of a cache has a dramatic influence on the performance of the binary heap. As long as the total size of the heap is small enough to fit into the caches, the operations are performed very quickly. Once the heap size increases beyond the cache size, the running time of the operations becomes significantly higher. This is due to the higher number of cache misses, which increases the time spent by the

memory system when loading and storing data. In other words, as long as the heap is small enough to fit into the cache system, the speed of the CPU is the main limiting factor. When the heap is much larger than the cache size, the speed of the memory system is the main limiting factor.

Bojesen [1] tested what happened when a binary heap exceeded the cache size when a heapsort of a certain number of elements was performed. In his graph over the time spent per element, the curve visibly breaks when the total size of the inserted elements is larger than what the cache(s) can hold. The heapsort begins to take a lot longer when the cache is full. He used 332 mhz and 160 mhz machines with 32KB and 16KB cache sizes. We tried the same test on a 800 mhz Pentium-III with 512KB level 2 cache. Due to the speed of the machine and the small first level cache size, it is not possible to get very accurate results around the number of elements where the size of the first level cache is exceeded. As the graph below shows, we did not get the same result as Bojesen, although there does seem to be a small change when the cache is exceeded (between 120000 and 140000 elements). Nevertheless, we are still convinced that it is a good idea to design algorithms to make



2.3 Cache Design

To be able to improve on the priority queue's cache-usage, we first need to know what a cache typically looks like and how it works. In this section we will provide a short description of the memory hierarchy. For a more

detailed reading we refer to [4].

Most modern computers have a 5 level memory hierarchy. When data is needed by the CPU, and these data are not in its registers, it will go to the next level in the hierarchy to search for these data, and so forth until the data is found. The lower it has to go in the hierarchy, the longer it will take to fetch the data.

The first memory level is the registers. Usually there are very few registers available, typically from 8 to 32. Register memory is very fast, and the CPU can normally read from and write to several registers in a single clock cycle.

The second level is the first level cache. This is a small chunk of very fast memory located on the same chip as the CPU. Normally the first level cache is split into an instruction and a data cache. What we are interested in here is the data cache. Data caches in relatively new machines typically have a capacity of 16KB to 64KB. This memory is divided into cache lines which can hold several machine words. Whenever the CPU requests a word which is not already in the cache, an entire cache line containing this word, and some of its nearest neighbours, is loaded into the cache. This is done because it is likely that the program will soon need the nearby data as well. It is probably also done, because the bus is wide enough to transfer this line of data, just as fast as it would transfer a single machine word. Of course it is best if the programmer and the compiler writer are aware of this locality principle and write their programs in such a way that they can benefit from it. Reading data from the first level cache takes 1-2 clock cycles.

The third level in the hierarchy is the second level cache. This cache works in much the same way as the first level cache, but it is much larger. Typically between 256KB and 2048KB. Also, it is located off the CPU chip, so accessing it takes longer than accessing the first level cache. Typical access times are 3-10 clock cycles.

The fourth level is the main system memory, usually called the RAM. This memory is slow but has a high capacity. Typically the size is from 32MB to 256MB on personal computers. Access times range from 40 - 100 clock cycles.

The fifth and lowest level is the secondary storage medium, typically a hard disk. Hard disks are very slow but have a very high capacity. Typically the size is from 8GB to 60GB on personal computers, but the CPU may not be able to access that much as memory. An address space of around 8GB is more common. Access times range from 700,000 to 6,000,000 clock cycles. Secondary storage is only part of the memory hierarchy if the operating system used supports virtual memory. It is the operating system's responsibility to swap pages between main system memory and the hard disk. A few older operating systems, without support for virtual memory, are still in limited use.

In our priority queue test cases we will try to avoid virtual memory page swaps, by not using more storage space than the RAM can handle. We do so because hard disk access times can vary greatly, even on the same machine, due to factors such as different seek times to different parts of

the disk surface and fragmentation of the files stored on the disk. For this reason, we feel that any test involving virtual memory page swaps, would give too varied and unreliable results to be useful.

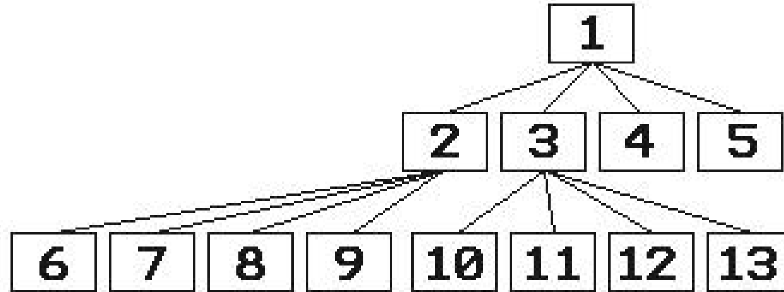
2.4 d-heaps

As the cache is so important for the performance of the heap, we would like to find an algorithm which makes better use of the cache. d-heaps, as described in [3], is an attempt at this. D-heaps are trees much like the traditional heap, where the d is a variable value representing the number of children of each node in the heap. A 2-heap is therefore exactly the same as the traditional binary heap. We want to see what happens when we use more than 2 children per node.

When we analyse d-heaps, we notice a CPU-time improvement when we have more children. The bubble operations performed when deleting the root node in the tree, examine for each node they go through, which of the node's children is the smallest. With more children per node, we get a sort of loop unrolling. Since the tree is shorter, for instance half as high when d = 4, the number of bubble operations are halved as well. The number of comparisons per bubble operations is doubled though, but these comparisons can be written as one line of code each. In other words, we have nearly halved the number of steps the bubble down loop goes through, while not introducing any new loops. We say "nearly", because this is only true for a worst-case delete min operation. The worst case is if the replacement for the root element has to be bubbled all the way to the bottom of the heap, which requires ($\log_2$ n) bubble operations in a binary heap. However, in most cases it will in fact go to either the lowest or the second lowest level, so on average the delete minimum operation requires close to ($\log_2$ n) bubble operations.

Anyway, using d-heaps with d $\geq$ 3 ought to reduce the number of instructions executed by the CPU to perform a delete. Of course, that was not really what we were after, but it is an added bonus. What we mostly wanted was improved cache usage. Let us see how d-heaps use the cache.

We assume that the d-heap is implemented in such a way that all the elements are placed in one consecutive block of memory. The elements are located in memory with the root node first, then its children, then its first child's children and so forth. Graphically a 4-heap looks like this:



Let us say we use a binary heap, one cache line holds 4 elements, and the root element is placed first in a cache line. In this case, every other sibling will be placed in two different cache lines. When comparing a node with its two children, and these are in different cache lines, we have to load both cache lines. Often we will not be using the three other elements in the second line, so of the memory work is wasted. We can improve on this situation by placing the root element at the end of a cache line. Then all siblings will be in the same line. That way we never waste more than of the memory work, because we always use at least 2 of the 4 elements in the cache line. However, we can do even better than that. This is where the d-heap comes in. If we use a 4-heap, a set of siblings will always fill out one cache line completely. That way we never waste any of the data we load into the cache.

2.4.1 Queue building choices

When a priority queue is created, one may want to build an initial queue from a list of elements, rather than insert each element separately. The constructor of the STL priority queue allows this to be done. We have two ways in which we can do this: Either we simply use the insert operation for each (called repeated-adds), or we use a special algorithm. The best known heap-building algorithm is Floyd's which runs in linear time. The tests performed in [3] shows that Floyd's algorithm executes fewer instructions than repeated-adds in a binary heap. However, Floyd's algorithm cannot take advantage of data locality as well as the repeated-adds can. Because of this, and because d-heap insertion runs in amortized constant time, Floyd's algorithm is only slightly faster when the heap fits in the 2nd level cache, and much slower when the heap size exceeds the cache.

Repeated-adds is faster when the heap size is large, and our aim is mainly to be fast with large heaps. Secondly, repeated-adds is very simple to implement compared to Floyd's algorithm. For these reasons we have decided to use repeated-adds for building the initial priority queue, not just in the d-heap case, but in all our implementations.

2.4.2 STL problems and solutions

The STL standard for priority queues, demands that all the elements that are inserted into the queue, are placed in a container of a user supplied class. We have no control over how this container allocates memory. Further more, the container does not allocate all the required memory at once, but allocates additional memory whenever it runs dry. This re-allocation probably requires that all the elements in the container are copied to a new location. At least this is true for the default vector container.

Secondly, cache-alignment requires knowledge of the cache line size. Since CphSTL can be used on many different machines, we would need some sort of configuration scheme which could provide us with this information. We do not at this time know if such information will be available to the CphSTL functions.

It would be possible, and perhaps interesting, to implement a special version of the vector container class, which ensures cache alignment of heap data. However, only the very observant and speed craving users of Copenhagen STL would notice and use such a container. We have decided to leave implementation of such a container as an exercise for the CphSTL project group working with vector. Our implementation of d-heap will not take advantage of cache alignment. Of course that will reduce the benefit of using d-heaps, because we have to be a bit lucky with the memory allocation, before it really pays off. Nevertheless, we still expect 4-heaps and higher grades to be faster than the binary heap.

Our implementation of the d-heap is based on the one presented by Bojesen in [1]. We have just changed the code to fit our as well as STL's requirements.

2.4.3 Pre-allocation of memory

The container for the heap elements will need to allocate additional memory now and then. Re-allocation takes time, so as far as possible it is probably best to avoid it. To try this, we have made a second version of the d-heap based priority queue. This second version allocates 512KB of memory for the container, at the time where the priority queue is constructed. If the priority queue used is significantly smaller than 512KB, this is of course a waste of memory. Also, this will only have an effect on the initial insertions. Once more than 512KB has been inserted in the queue, the pre-allocation has no effect any more. In other words, if the job performed with the priority queue progresses over a long period of time, the speed gain will seem very small compared to the total running time. However, for short operations such as a heapsort, where you just insert and then extract, it ought to lower the running time noticeably.

We choose the value 512 KB, because it seems like a good compromise. It should be enough to have an effect, but at the same time the possible memory waste is within reasonable bounds.

2.5 The ranked priority queue

In earlier studies, priority queues implemented with the use of pointers have shown to be the most efficient algorithms. According to Lamarca and Ladner this is no longer true, because pointer based algorithms spread related data to different locations in memory. That means that they do not exploit the cache very well, because usually only a part of a loaded cache line will be used. Further more, pointer based algorithms use more memory, because of all the pointers that need to be stored along with the actual data. In modern machines memory is much slower than the CPU, so an algorithm requiring many calculations but few data, will normally be faster than one requiring few calculations but a lot of data. Pointer based priority queues may be fast when the cache can hold all their data. However, due to the large amount of memory required to store the pointers, the cache will be filled more quickly (measured in number of elements inserted) than when using a binary heap or d-heap.

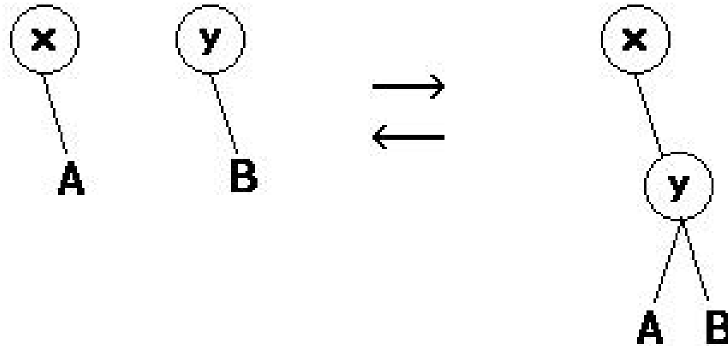
Of course we would like to see these results for ourselves. For that reason we have tested a pointer based priority queue known as the "ranked priority queue". This algorithm is described in additional detail in [2].

The ranked priority queue has the advantage that it can perform insertion in constant time. This sounds like an improvement over the binary heap algorithm, which has a worst case time for insertion of $O(\log_2 n)$. However, this worst case only arises when the inserted element is a new minimum, which has to be bubbled all the way to the root. Studies have shown that on average only 1.6 bubble operations are performed on each insertion, provided that the key values of the elements are uniformly distributed [3]. Apparently this is true even for large heaps. Knowing this, the promise of constant time insertion sounds less intriguing.

The ranked priority queue makes use of heap trees. To define these, we first need to define half-ordered trees. A **half-ordered tree** is a binary tree where for any node v, v has key value less than or equal to the key value of any node in v's right subtree. Now we can define a heap tree. A **heap tree** is a half-ordered tree, where the root has at most one child, which is then a right child.

In this algorithm we will use what is called a leftmost path. A **leftmost path** is a path in which all edges go to left children and in which the last node has no left child. We also use two relatively simple operations called linking and un-linking. One is the direct opposite of the other. A link combines two heap trees into one heap tree the following way: The root with the smallest key value becomes the root in the linked tree, the root node of the other tree becomes its right child. The rest of the first tree becomes the left subtree of the this node, while its own right subtree remains in place. The un-link takes a heap tree and splits it into two by doing the directly opposite. This is shown graphically below. x and y are root nodes while A and B are their subtrees. The arrow right leads to the result of linking these two. The arrow

left leads to the result of un-linking the tree on the right.



As this is called a ranked priority queue, a ranking scheme is of course involved. Each node in the heap trees used for the priority queue, has a rank. The rank must be within these bounds:

For any node v in a heap tree:

1. if v is a root with no child, $rank(v) = 1$
2. if v is a root with a right child, $rank(v) = rank(right(v)) + 1$
3. if v is a non-root node, $\mathrm{l}(v) \leq k * rank(v) \leq \log_{(1/a)}(\mathrm{s}(v) + 1)$

Here $\mathrm{s}(v)$ is the number of nodes in the tree rooted at v, $\mathrm{l}(v)$ is the length of the leftmost path from v, and $\mathrm{right}(v)$ is the right subtree of v, $\frac{1}{2} \leq a < 1$ and $k = 1$. When decreasekey and other advanced operations are required by the priority queue, an advanced ranking system is necessary to preserve balance in the heap trees. However, with the operations we need to implement in the STL priority queue, we can make do with a very simple system. We set $k = 1$ and $a = \frac{1}{2}$. With these values, the rule for leaf nodes says that: $1 \leq 1 * rank(v) \leq \log_2(1 + 1)$ which reduces to $1 \leq rank(v) \leq 1$. In other words the rank of a leaf node is always 1. Since the only operation we perform on the heap trees are linking and un-linking, all subtrees below the root will be balanced. This means that $2^{l(v)}$ - 1 = $\mathrm{s}(v)$ so our equation becomes $\mathrm{l}(v) \leq 1 * rank(v) \leq \log_2(2^{\mathrm{l}(v)} - 1 + 1)$ which reduces to $\mathrm{l}(v) \leq rank(v) \leq \mathrm{l}(v)$. In other words, the rank of a non-root non-leaf node is always one greater than the rank of its child nodes.

Now we are ready to describe the algorithm for the ranked priority queue: We will be using two lists. The first one is called Q. This is a list where each entry is either empty or contains one heap tree. Entry one (Q_1) contains a heap tree of rank one, entry two (Q_2) one of rank two and so forth. The other list is the tree list. The tree list contains a number of pairs of heap trees. The two trees in each pair have the same rank, but the pairs are placed in the tree list in no particular order. Finally we also maintain a pointer to the node containing the smallest node in the queue.

Insertion is done in two steps:
1. We create a new node with rank 1. If Q_1 is empty, then this node is inserted in Q_1. If a tree is already in Q_1, we create a pair containing the new node as well as the node in Q_1. This pair is inserted in the tree list and Q_1 is set to empty. We also check if the inserted node is a new minimum. In that case we change the minimum node pointer to point here.
2. We now check if the tree list is empty. If it is not, we remove the first pair in the list and link the two trees it contains into one. This tree has rank r. If Q_r is empty, the tree is inserted into Q_r. Otherwise, we create a pair containing this tree and Q_r, and insert this into the tree list, while Q_r becomes empty.

Although the tree list and the Q list grow in size over time, they can be allocated when the priority queue is created. After all, it is unlikely that computers within the next hundred years will be able to address more than 2^{100} bytes of memory. As these two lists will never be longer than $\log_2$ n, where n is the number of elements in the queue, pre-allocating 100 entries should be more than enough. Even if Copenhagen STL is still in use in 100 years, updating the code will only be a matter of changing a single constant.

Deletion of the minimum node is a bit more complicated:
1. As we have a pointer to the minimum node as well as its rank, we can quickly check if the tree having this node as its root, is in the Q list. If it is in the Q list, we remove this tree from its entry in Q which then becomes empty. If the tree is in the tree list, we have to search through this list to find it. We then remove this pair, and reinsert the other half of the pair by the same principle as when we insert single nodes.
2. We now hold the tree with the minimum node. Lets call this tree T. If T has a right subtree, we un-link T. The second tree resulting from this operation (the one not containing the minimum node) is reinserted in the queue by the same principle as when inserting new nodes. If the first tree has a right subtree we repeat this un-link process until we only hold the minimum node. This is then deleted.
3. Finally we look through all the trees in Q and in the tree list to find the new minimum node and update the minimum node pointer.

2.5.1 STL problems and solutions

As stated earlier, the STL standard requires that all elements in the queue are placed in a container of a user supplied class. As this algorithm makes use of pointers, it would be very difficult, if not impossible, to store the elements in such a container. For this reason, our implementation does not use the supplied container class, and therefore does not fully adhere to the standard.

2.6 The external priority queue

The external priority queue is based on The External Heapsort as it was presented by Wegner and Teuhola in [6]. We refer to that article for a more detailed description. The external heapsort was meant to be used with large heaps that had to be partially stored on disk or other forms of secondary storage. It becomes useful for our purpose, if we consider the cache to be the primary storage space, while the RAM is the secondary storage. The way the external heapsort swaps pages between primary and secondary storage, may turn out to be a good way of ensuring a high degree of data locality, leading to high cache performance. Fortunately it is possible to turn the external heapsort into a priority queue algorithm.

The external priority queue divides the data into a number of pages of equal size. Most of these pages reside in external storage, while only 4 of them need to be in core storage at any given time. The pages are ordered as nodes of a binary heap. To be able to ditinguish one from the other, we will call this page heap a hill instead. The hill has the property that no element in a child page has a key value which is smaller than the key value of an element in the parent page. This is very useful for sorting, because during extraction of the elements, an entire page can be removed at a time.

In the priority queue case, we are dependent on how the user interleaves adding and removal of elements. The priority queue will therefore use an insertion buffer and an extract buffer. These are each one page large. Whenever a new element is inserted, it is placed in the insert buffer. The insert buffer is a normal heap structure. We will be using 4-heaps for our tests. With this buffer in use, most inserts will just require the normal d-heap insertion procedure. Only when the insertion buffer is full is access to external pages necessary, as the insert buffer will then be transfered to a page which is inserted in the hill. This requires that all the elements in the d-heap are deleted and in addition one hill merge operation and one hill bubble up operation.

The extract buffer works much the same way. When a priority queue element is deleted, we check if both the insert buffer and the extract buffer are empty. In that case we take a page from the hill and transfers it to the extract buffer. This requires two page copying operations as well as a bubble down operation in the hill. If the extract buffer is empty, but the insert buffer is not, we can handle a priority queue delete operation by taking the top element from the insert buffer. When both buffers have elements in them, we compare the top elements, and removes the top from the buffer which holds the smallest element.

The hill bubble operations merge and copy pages from the hill. For details on merging and bubbling algorithms, we refer to [6] where they are shown as Pascal code. However, as noted before, no more than 4 pages need to be in core memory at any time. In our tests, we will consider the first level cache the core memory. Each page will therefore have a size of $\frac{1}{4}$ of the first

level cache size.

2.6.1 STL problems and solutions

Our implementation of the external priority queue is based on the one presented by Bojesen in [1]. We have just changed the code to fit into STL's priority queue class. Bojesen's implementation pre-allocates all the memory needed for the external storage block. It is possible, but not easy, to rewrite the code so that it instead makes use of the user supplied container class, and allocates memory as needed. However, the code will certainly not be any faster if we do this. To save us some trouble, we will first test if the external priority queue is the fastest method when using an implementation close to Bojesen's. If this is not the case, there is no reason why we should rewrite the code to fit STL perfectly.

3. The benchmark tests

To test the different implementations of the priority queue, we have constructed the following benchmark test programs:

3.1 PQSort

PQ-sort is the same as a heapsort, we just use a priority queue instead of a heap.

We have used this method to sort elements of three different sizes: 4 bytes, 20 bytes and 32 bytes. All elements are objects containing a key value used for sorting them, as well as an array with random values, which is only used to fill out the object to a certain size. These fill bytes are never read from or written to by the program. The 20 byte structure was used for two reasons. Firstly, a number of these objects probably will not fit into one cache line. Caches normally use cache line sizes which are a power of 2. It is interesting to see how this affects the algorithms that directly attempt to exploit the cache. Secondly, the ranked priority queue adds 12 bytes to each element (assuming its a 32 bit machine). With 20 byte elements, all elements become 32 bytes long, which should fit into a cache line. If the ranked priority queue does not outperform the other algorithms in this highly biased case, surely it never will.

With a 4-byte element size, we have run this sorting test with 100,000 ; 500,000 ; 1,000,000 ; 1,500,000 and 2,000,000 randomly generated elements. However, for the ranked priority queue we have gone no higher than 500,000 elements, due to limitations on available RAM.

With a 20-byte elements size, we have run the test with 100,000 ; 200,000 and 300,000 elements.

With a 32-byte elements size, we have run the test with 100,000 ; 150,000 and 200,000 elements.

3.2 Dijkstra's algorithm for finding shortest path trees in a graph

This algorithm takes a directed weighted graph, starts at some node N, and creates a tree from the graph. The tree is rooted at N and contains the shortest path from N to all the other nodes in the graph which are reachable from N. We will not go into details on how the algorithm works, but just say that while generating the tree, the algorithm always looks at the marked node with the smallest current distance value. As nodes will be marked, unmarked and change their distance value while the algorithm runs, it is obviously a good idea to use a priority queue to sort the marked nodes. The node we need to process next will always be at the top of the queue. Our implementation of the algorithm is based on the one presented in [5].

To be able to test the algorithm on various graphs of different sizes, we have created a graph generator program. The generator can generate graphs where all nodes are reachable from the first node in the graph. Each node can have 0 to 4 outgoing links to other nodes, but any number of ingoing links. We need two machine words to store each link, one to hold the number of the node linked to, another to hold the cost of that path. With a maximum of 4 outgoing links, each node in the graph can be stored in an array, where each node's entry is 32 bytes large (if it is a 32 bit machine).

We have run this test with 100,000 ; 200,000 and 300,000 nodes in the graph. The elements inserted into the priority queue are 8 bytes in size.

3.3 LongPQTest

This test does nothing useful, it just tests three aspects of the priority queue. First it inserts 500,000 4-byte elements, then it deletes the top element 100,000 times. It then inserts another 500,000 elements, then deletes the top element 50,000 times while first reading the top element and assigning it to a variable (some compilers may choose to remove this as the variable is not used elsewhere). Finally it inserts 500,000 elements, while deleting the top element once every time it has inserted 100 of them.

3.4 InsertTest

The insert test just inserts a number of elements. This test is mainly used to see if the ranked priority queue's constant time insertion makes it faster in this respect. We have tested this with insertion of 100,000 and 300,000 elements with a size of 4 bytes and a size of 20 bytes. We have also run this test with insertion of 100,000 and 200,000 elements with a size of 32 bytes. Once again we would expect the ranked priority queue to work best with 20 byte elements.

For every test above, the values that have been used as key values for the elements, were generated randomly using C's rand() function. We generated

values like this:

```
keyval = (rand() % 32700) * (rand() % 32700)
```

It was done this way, because the range of values that `rand()` can return, varies from compiler to compiler. It is however unlikely, that their maximum value would be less than 32700. By restricting the range of values, we ensure that the test result is independent of the `rand()` implementation.

3.5 Test machines

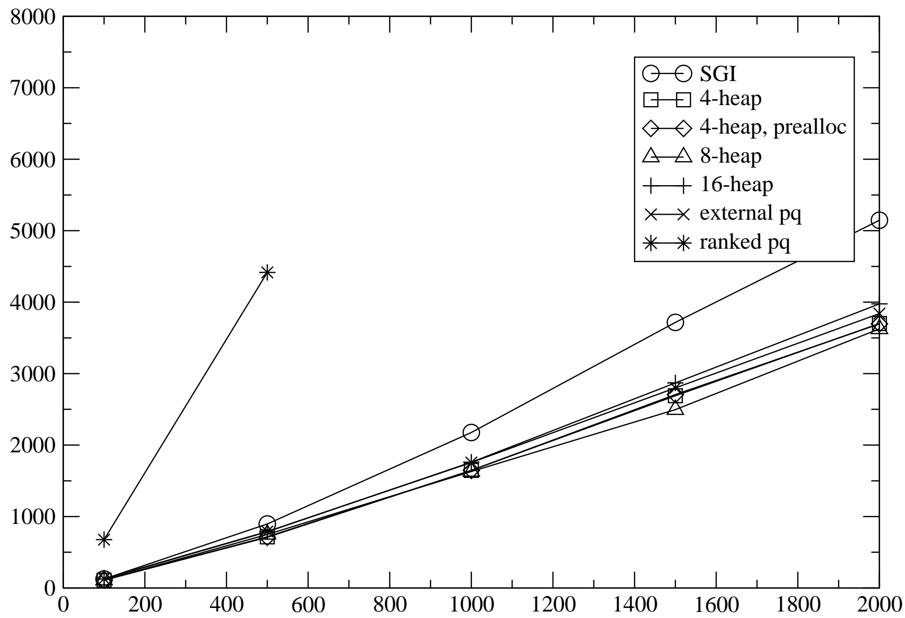
The test programs were run on 4 different machines. Unfortunately we were not able to obtain all the interesting information about the hardware on these machines. Uncertain and unknown data about the machines are marked with a '?' in the table below.

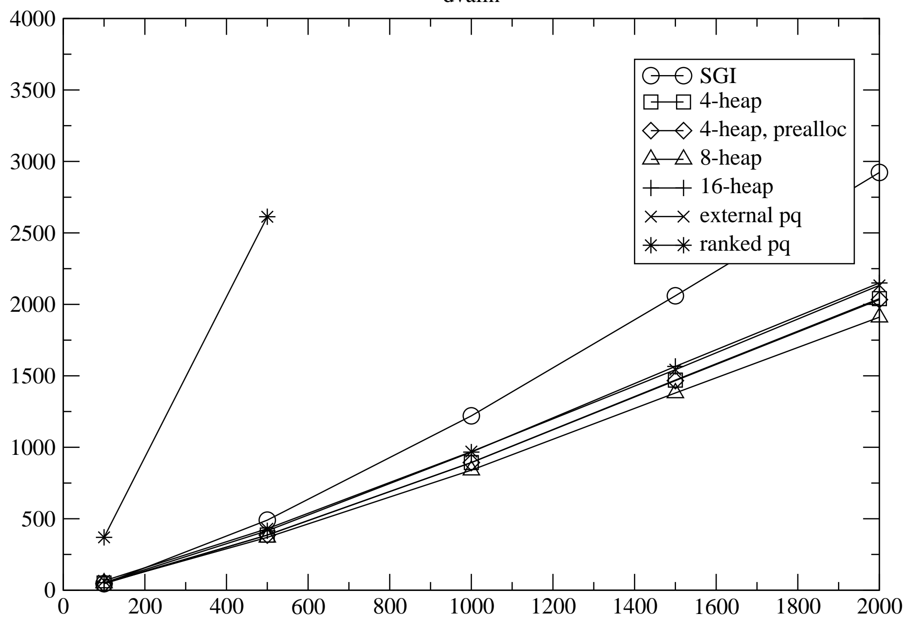
DIKU name	**CPU**	**L1 cache**	**L2 cache**	**RAM**
private machine	Pentium 3 - 450 mhz	32 KB?	512 KB	64 MB
dvalin	Pentium 3 - 800 mhz	32 KB?	512 KB	1 GB
heimdal	(HP 9000/C160 ser.)	?	?	80MB?
thor	PA 7100 - 99 mhz	?	256 KB	80 MB

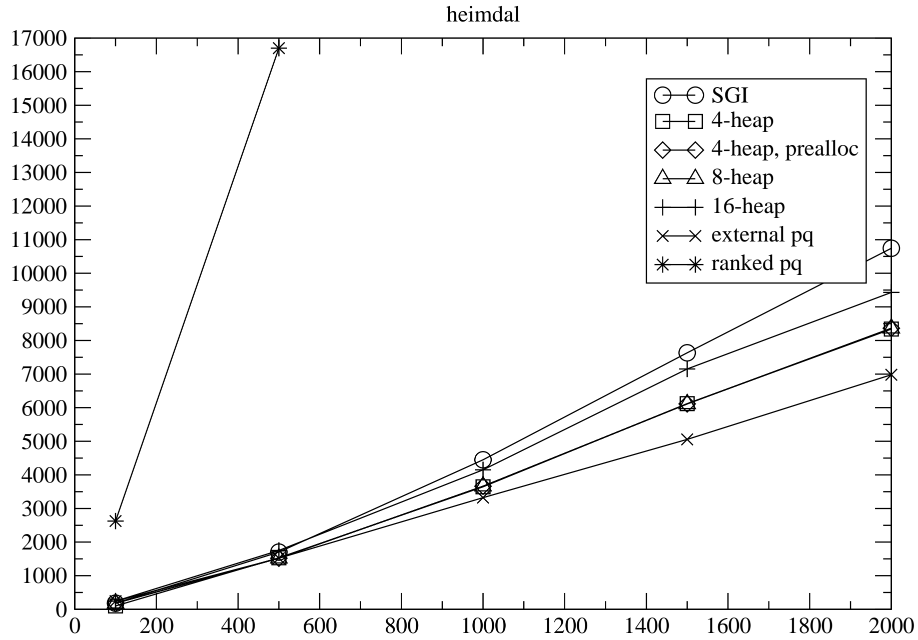
We ran through each test 3 times on every machine, while the machines were as unloaded as possible. The results presented below are averaged over these three runs. The times on the y-axis are measured in milliseconds.

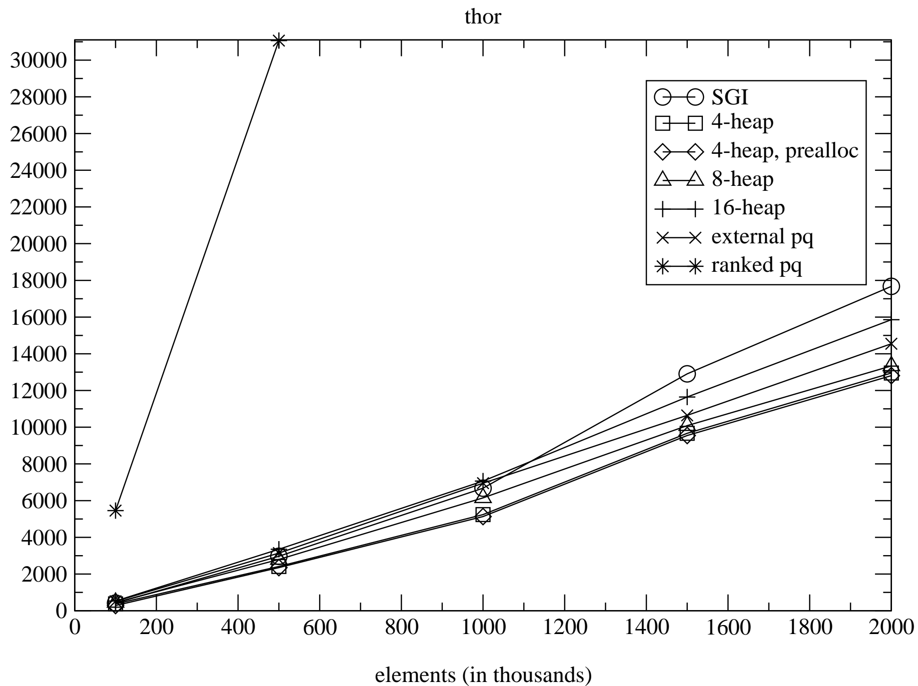
Since we were not certain on some of the cache sizes, we used a page size setting of 8KB on all machines when testing the external priority queue. The 4 needed in core memory then takes up 32KB.

3.6 PQSort test results





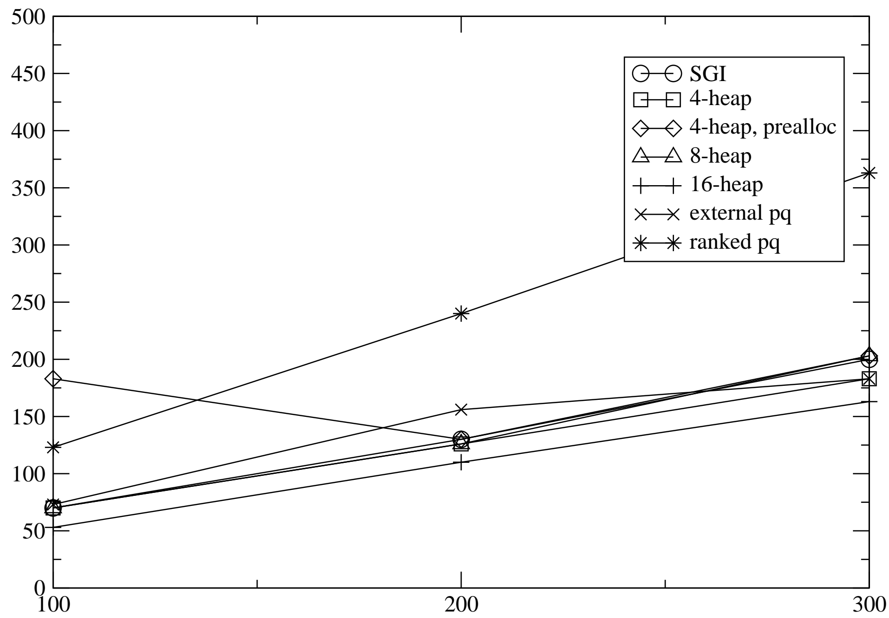


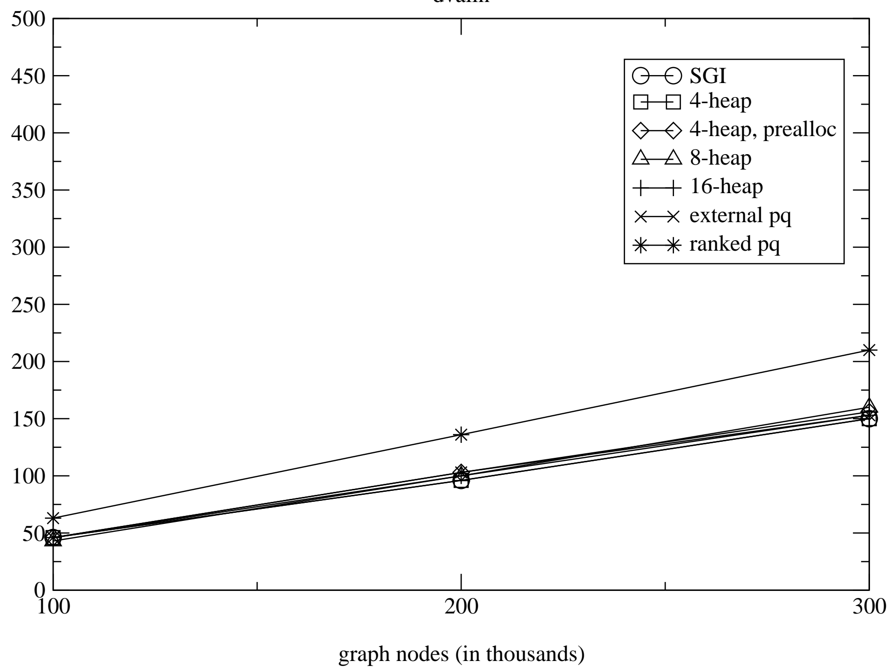


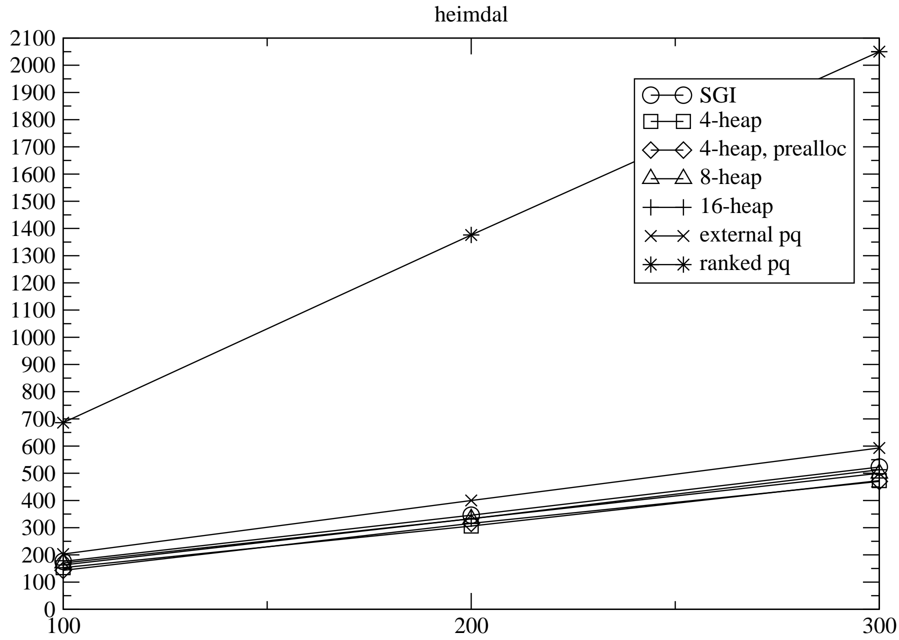
Let us have a look at the PQSort results for the two Pentium-III machines. These two machines have the same cache sizes but one has a 450 mhz processor, the other a 800 mhz processor. It is interesting to see that in the

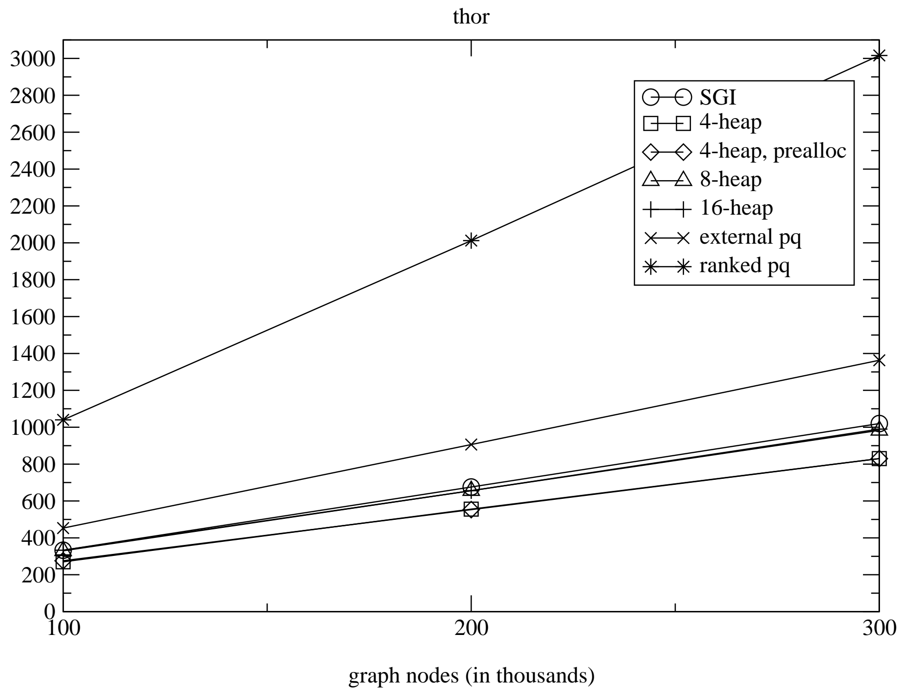
case with 100,000 elements of 4 bytes, where the entire heap can be stored in the caches, the 800 mhz machine is 2.74 times as fast as the 450 mhz machine, when looking at SGI's implementation of the priority queue. That it is more than twice as fast, may be because the compiler used on the fast machine is better at optimizing the code. Another possibility is that the system bus and RAM on the 800 mhz machine is faster. We have not been able to obtain this information about the hardware. However, in the case with 500,000 elements, the 800 mhz machine is now only 1.83 times as fast, and in the case with 2,000,000 elements it is 1.76 times as fast. Clearly the speed of the memory system has more to say in the latter cases.

3.7 Dijkstra test results









Dijkstra did a very good job on this algorithm. It is actually a bit too fast for our purpose. Since most of the work is not done by the priority queue, the running time difference between the 5 algorithms does not show up as

clearly.

What we do see however, is the the d-heaps are faster than the binary heap on all machines except for dvalin, where they are pretty much the same. The external priority queue comes close on all machines except thor where it is significantly slower. The ranked priority queue is definitely last on all machines.

3.8 Additional test results

Finally we present some of the results obtained with 20 and 32 byte element sizes. As the results are quite similar on all machines, we have just picked out a few.

PQSort with 20 byte elements on *private machine*:

	100K elements	**200K elements**	**300K elements**
SGI version	403 ms	990 ms	1596 ms
4-heap	423 ms	1153 ms	1480 ms
4-heap prealloc.	380 ms	873 ms	1430 ms
8-heap	386 ms	860 ms	1376 ms
16-heap	383 ms	880 ms	1426 ms
external p.q.	600 ms	1370 ms	2160 ms
ranked p.q.	806 ms	1810 ms	2930 ms

PQSort with 32 byte elements on dvalin:

	100K elements	**200K elements**	**300K elements**
SGI version	370 ms	610 ms	836 ms
4-heap	346 ms	580 ms	770 ms
4-heap prealloc.	346 ms	570 ms	770 ms
8-heap	310 ms	510 ms	680 ms
16-heap	316 ms	520 ms	693 ms
external p.q.	543 ms	890 ms	1230 ms
ranked p.q.	506 ms	830 ms	1186 ms

InsertTest with 20 byte elements on thor:

	100K elements	**300K elements**
SGI version	483 ms	1726 ms
4-heap	470 ms	1670 ms
4-heap prealloc.	430 ms	1036 ms
8-heap	480 ms	2136 ms
16-heap	476 ms	2126 ms
ranked p.q.	590 ms	1756 ms

LongPQTest on *private machine*:

SGI version	803 ms
4-heap	716 ms
4-heap prealloc.	713 ms
8-heap	676 ms
16-heap	676 ms
external p.q.	1756 ms

LongPQTest on heimdal:

SGI version	2413 ms
4-heap	2143 ms
4-heap prealloc.	2096 ms
8-heap	2533 ms
16-heap	2596 ms
external p.q.	3323 ms

As expected the ranked priority queue performs quite well with the highly biased 20 byte elements. On dvalin it was actually faster at inserting than both SGI's and our d-heap implementations. On heimdal and thor it was a lot slower though. When it comes to sorting however, the ranked priority queue can no longer compete, due to its relatively slow delete operation.

With the 32 byte elements, the ranked priority queue also ought to perform well, but it only marginally beats the external priority queue, while being slower than all the others.

The external priority queue performs badly with the large element sizes. It is far behind the binary heap and the d-heap methods. In the cases with 4-byte elements, the external priority queue was almost on par with the d-heap. It may be because each page contains very few elements when the element size is so large.

On the LongPQTest, the external priority queue is far behind as well. This may be due to the last section of the test, where deletions and insertions are interleaved.

From our insertion tests, we see that the d-heap with preallocation is faster than the d-heap without pre-allocation. This is as we expected. Somewhat surprisingly, its head start does not seem to pay off in the long run. In many of the PQSort and Dijkstra tests it ran at the same speed, sometimes it was even slower. It may be that the copying performed by the container during allocation of more memory, is in fact beneficial to the cache performance later on.

4. Conclusions

It is our recommendation that the 8-heap functions and the 8-heap based priority queue should be the ones used in Copenhagen STL. Our tests have shown that the d-heap algorithm is the fastest in almost every case. We recommend the version with a fanout of 8, as this seems to give the best performance on most machines in most of the tests. The difference between 4 and 8 is small but noticeable. On the older machines, the 4-heap is of-

ten faster than the 8-heap. However, on the newest machines the 8-heap is fastest, so that should be chosen. A fanout of 16 does not seem to be beneficial. In most cases this was slower than both 4 and 8. Also, pre-allocation of memory is not recommendable. Sometimes it has a small positive influence, sometimes a small negative influence. The difference is rarely any larger than what can be attributed to statistical uncertainty.

The external priority queue often comes very close to the d-heap, in a few cases it even performed better. We have only tested this algorithm using a 4-heap insertion buffer and with a page size fixed at 8KB. It may be worthwhile to try with an 8-heap as well. As the algorithm is promising, a future project should be to test it with an even greater number of elements inserted in the queue, to see how it performs when virtual memory pages are swapped to secondary storage. One could also try to improve a bit on the algorithm or the implementataion of the page swapping and merging. Perhaps something is to be gained here. A problem with the external priority queue is however, that it is not easily changed to fit the STL requirements.

Finally our tests have delivered another piece of evidence that pointer-based priority queues are not the way to go. They simply use too much memory and cannot properly take advantage of the cache. Even on the oldest machine with a relatively slow 99 mhz processor and only 256 MB 2nd level cache, the ranked priority queue was extremely slow.

5. Bibliography

[1] Bojesen, J. (1999), *Heap implementations and variations*, http://www.diku.dk/research-groups/performance-engineering/Jesper/heaplab/heapsurvey_html/Welcome.html.

[2] Høyer, P. (1994), *A General Technique for Implementation of Efficient Priority Queues*, http://www.brics.dk/~hoyer/papers/pq94preprint.ps.gz.

[3] LaMarca, A. & Ladner, R. E. (1997), "The Influence of Caches on the Performance of Heaps," In *The ACM Journal of Experimental Algorithmics.*

[4] Hennessy, J. L. & Patterson, D. A. (1996), *Computer Architecture: A Quantitative Approach*, second edition, Morgan Kaufmann Publishers, Inc.

[5] Liljedahl, E. & Christensen, O. K. (1995), *Datalogi 1P Kursusbog 1994-1995: Grafalgoritmer og kompleksitet*, DIKU Tryk.

[6] Wegner, L. M. & Teuhola J. I. (1989), "The External Heapsort," In *IEEE Transactions on software engineering vol 15*, p917-925.