

## Assignment 5: Priority Queue

---

Thanks to Julie and Mehran for polishing off this assignment.

So, it's finally time for you to be implementing a class of your own: a priority queue, which is a variation on the standard queue described in of the reader. The standard queue is a collection of elements managed in first-in, first-out ("FIFO") manner. The first element added to the collection is always the first element extracted; the second is second; so on and so on.

In some cases, a FIFO strategy may be too simple for the activity being modeled. A hospital emergency room, for example, needs to schedule patients according to priority. A patient who arrives with a more serious problem should pre-empt others even if they have been waiting longer. This is a *priority queue*, where elements are added to the queue by priority, but when time comes to extract the next element, it is the highest priority element in the queue that is removed. Such an object would be useful in a variety of situations. In fact, you can even use a priority queue to implement sorting; insert all the values into a priority queue and extract them one by one to get them in sorted order.

The main focus of this assignment is to implement a priority queue class in several different ways. You'll have a chance to experiment with arrays, linked lists, and chunk lists (more on those later). Once you have debugged your implementations, you will run some tests and consider the strengths and weaknesses of the various versions. We provide client code that tests and times the performance of the class, while your role will be to act as the implementer.

**Due: Monday, November 2<sup>nd</sup> at 11:00 a.m.**

### The PQueue Interface

The priority queue will be a collection of integers, where the integer itself is used as the priority. Larger integers should be considered *higher* priority than smaller ones and, thus, removed before smaller values. Here are the functions that make up the public interface of the priority queue:

```

class PQueue {

public:
    PQueue();
    ~PQueue();

    int size();
    bool isEmpty();
    void enqueue(int elem);
    int extractMax();
    // some additional methods for compiling metrics

private:
    // implementation dependent member variables and helper methods
};

```

**enqueue** is used to add a new element to the priority queue. **extractMax** returns the value of highest priority (i.e., largest) element in the queue and removes it. For the detailed specification of the behavior and usage of these functions, see the **pqueue.h** interface file included in the starter files.

### Implementing the priority queue

There are many data structures you could choose to represent and manipulate a priority queue, with various tradeoffs between the amount of memory required, speed of the insert and extract operations, complexity of code to get the job done, etc. The first implementation will store the priority queue elements in an unsorted array. The second implementation will represent the queue as a sorted linked list. The third implementation is a hybrid cross between an array and a linked list, a *chunklist*. The fourth implementation will represent the priority queue as a *heap* (not to be confused with the heap where memory is dynamically allocated via **new**). The first two implementations are provided to you, but the last two will be yours to write.

### Unsorted **vector** implementation

The unsorted vector implementation is already written and given to you in the starter project. You do not have to write any code for this implementation, just go over and be familiar with the code provided. This implementation is layered on top of our **Vector** class. Its **enqueue** strategy is simply to add the new element to the end. When it comes time to **extractMax** the maximum element, it performs a linear search to find it. You will run time trials on this implementation as it stands to get an idea of its strengths and weaknesses and become familiar with the test and time trial functions we've provided for you.

### Sorted linked list implementation

The sorted linked list implementation is also given to you so you can observe and test it. The linked list implementation is a singly linked list of values, where the values are kept in

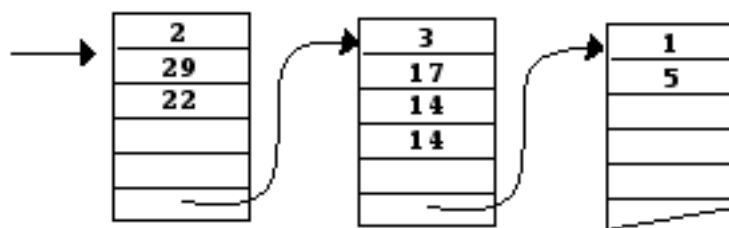
reverse-sorted (i.e., largest to smallest) order to facilitate finding and removing the largest element quickly. However, inserting a value is a bit trickier because of the need to search for the right position to insert the new value. You will also run time trials on this implementation as part of your analysis.

### Sorted chunklist implementation

Now it's your turn! Neither the array nor the linked list is a particularly strong performer in terms of space or time efficiency, so next you'll try concocting a combination of the two to see what advantages a hybrid might offer. We still maintain the sorted property of the original linked list but try to reduce some of the memory overhead and slow traversal time by making each cell not a single element, but a block of elements. Thus, the **chunklist** combines the array and linked-list modules so that the actual structure consists of a singly linked set of blocks, where each block contains a fixed size array capable of holding several elements rather than a single one.

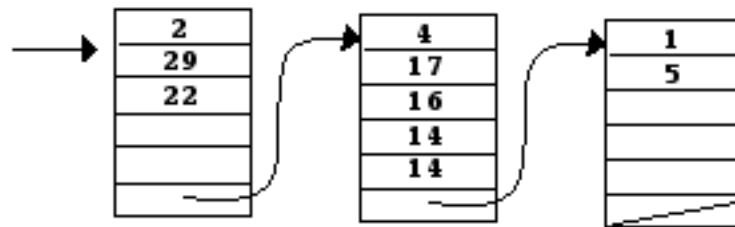
By storing several elements in each block, you reduce the storage overhead because the pointers take up a smaller fraction of the total size of a cell. However, because the blocks are of a fixed maximum size, inserting an element into a block never requires shifting more than  $k$  elements, where  $k$  is the **block size** or maximum number of elements per block. The time it takes to find the right block in which to insert a new element is also reduced by the added ability to step over entire blocks of elements, rather than examining each element one by one. The elements are still kept in reverse-sorted order, to facilitate an easy **extractMax** operation.

To get a better idea of how this new priority queue representation works, consider the diagram below for a priority queue with a chunk size of 4. Because the blocks need not be full, there are many possible representations for the same contents. For example, this priority queue containing 29, 22, 17, 14, 14 and 5 might be divided into three blocks, as follows (the first number in each block is the number of elements used in this block):

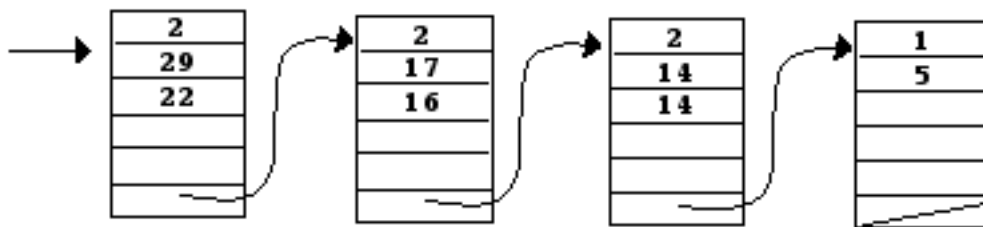


Suppose that you want to insert **16** and **15**. Inserting a **16** is a relatively simple matter because the appropriate block has only three elements, leaving room for an extra one. We shift the **14**s toward the end of the block, but do not need to make changes to the pointers

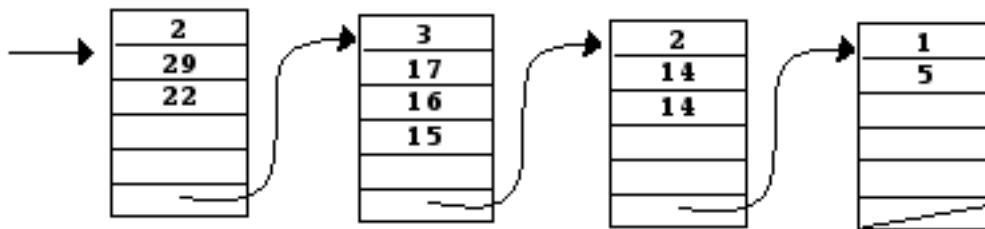
linking the blocks themselves. The configuration after inserting the **16** therefore looks like this:



If you now try to insert a **15**, however, the problem becomes more complicated. At this point, the current block is full. To make room for the **15**, you need to move some of the elements to another block. A simple strategy is to distribute the elements across two blocks: the current block and a newly allocated block. After splitting (but before inserting the **15**), the priority queue looks like this:



After splitting, it is a simple matter to insert the **15** in the appropriate block:



Your job is to implement the priority queue as a linked list of blocks using a general strategy like that shown above.

You must implement a *sorted* chunk list, but the detailed data structure decisions are yours to make. Think carefully about what is needed to support the operations efficiently. Avoid redundancy and complication, especially where it provides no benefit. Make sure that you have a clear understanding of how your data structure works before you start writing the code. Draw pictures. Figure out what the empty priority queue looks like. Consider carefully how the data structures change as blocks are split. The chunk size should be specified with a **MaxElemsPerBlock** constant (which could be odd, and as small as 2) and it should be possible to change that value to experiment with the different tradeoffs.

If you have to insert an element into a block that is full, you should adopt the strategy described above and divide the full block in half before making the insertion. This policy helps ensure that neither of the two resulting blocks starts out being filled, which might immediately require another split when the next element comes along.

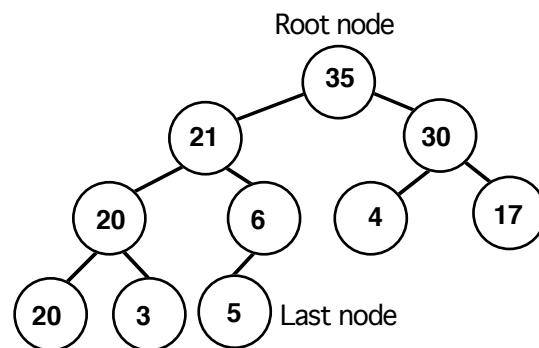
Extracting the largest element means removing the first element in the first block. You can decide whether it makes sense to shuffle down the elements in the rest of the block or use some technique to know what contents the block has. If you delete the last element in a block, your program should free the storage associated with that block.

## Heap implementation

Although the binary search trees we'll eventually discuss in lecture might make a good implementation of a priority queue, there is another type of binary tree that is an even better choice in this case. A *heap* is a binary tree that has these two properties:

It is a *complete* binary tree, i.e. one that is full in all levels (all nodes have two children), except for possibly the bottom-most level which is filled in from left to right with no gaps. The value of each node is greater than or equal to the value of its children.

Here's a conceptual picture of a small heap:

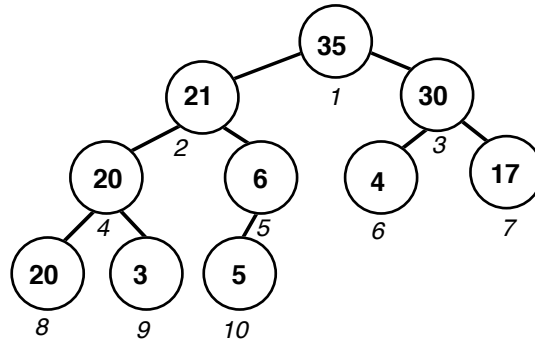


Note that a heap differs from a binary search tree in two significant ways. First, while a binary search tree keeps all the nodes in a sorted arrangement, a heap is ordered in a much weaker sense. Conveniently, the manner in which a heap is ordered is actually sufficient for the efficient performance of the priority queue operations. The second important difference is that while binary search trees come in many different shapes, a heap must be a complete binary tree, which means that every heap containing ten elements is the same shape as every other heap of ten elements.

## Representing a heap using an array

One way to manage a heap would be to use a standard binary tree node definition and wire up left and right children pointers to all nodes. We can exploit the completeness of

the tree and create a simple array representation and avoid messing with pointers. Consider the nodes in the heap to be numbered level by level like this:



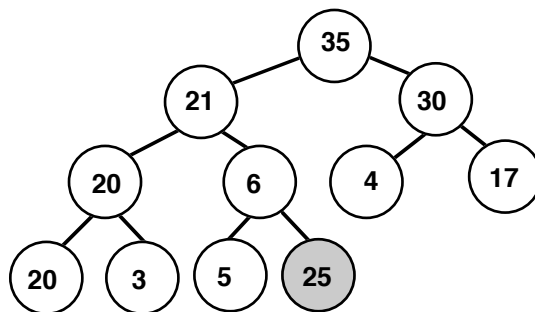
and now check out this array representation of the same heap:

35	21	30	20	6	4	17	20	3	5
1	2	3	4	5	6	7	8	9	10

You can divide any node number by 2 (discarding the remainder) to get the node number of its parent. For example, the parent of node 11 is node 5. The two children of node  $i$  are  $2i$  and  $2i + 1$ , e.g. node 3's two children are 6 and 7. Although many of the drawings in this handout use a tree diagram for the heap, keep in mind you will actually be storing the heap in an array.

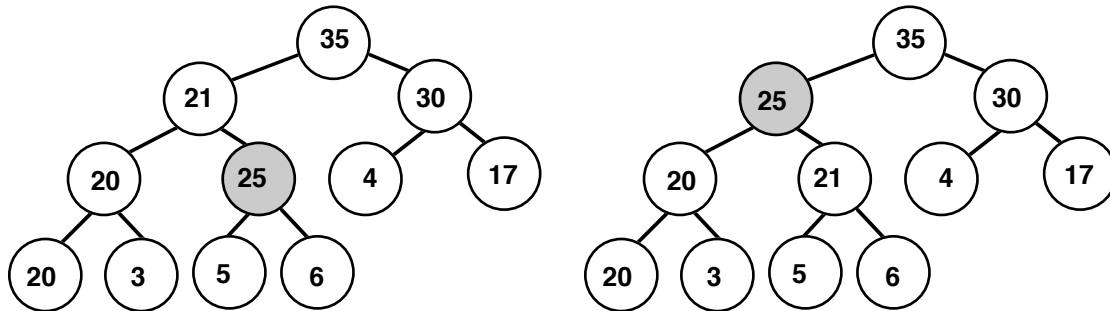
### Heap insert

Inserting into a heap is done differently than its functional counterpart in a binary search tree. A new element is added to the very bottom of the heap and it rises up to its proper place. For example, we want to insert 25 into our heap. First we add a new node at the bottom of the heap (the next position to add in the tree is dictated by the completeness property):



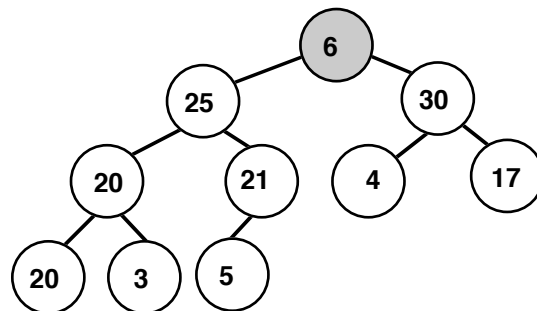
35	21	30	20	6	4	17	20	3	5	25
1	2	3	4	5	6	7	8	9	10	11

We compare the value in this new node with the value of its parent and, if necessary, exchange them. Since our heap is actually laid out in an array, we can move the nodes by swapping array values. From there, we compare the moved value to its new parent and continue moving the value upward until it needs to go no further. This is sometimes called the *bubble-up* operation.

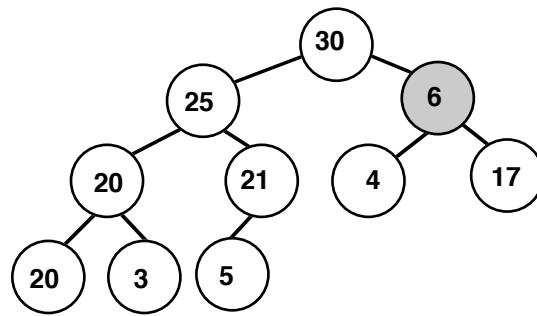


### Heap remove

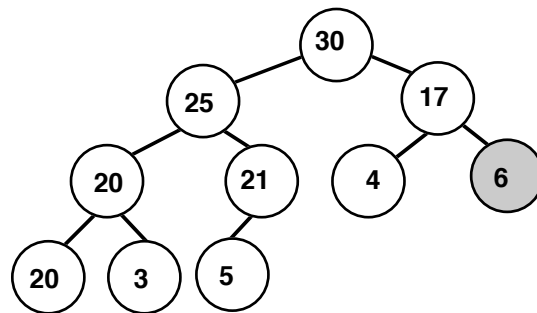
Where is the largest value in the heap? Given heap ordering, the largest value is in the root, where it can be easily accessed. However, after removing this value, you must re-configure the remaining nodes back into a heap. Remember the completeness property dictates the shape of the heap, and thus it is the bottommost node that needs to be removed. Rather than re-arranging everything to fill in the gap left by the root node, we can leave the root node where it is, copy the *value* from the last node to the root node, and remove the last node.



At this point, we have a complete binary tree again, whose left and right subtrees are heaps. The only problem is that the item in the root may be (and usually is) out of place. In order to restore the heap ordering property, we need to trickle that value down to the right place. We will use an inverse to the strategy that allow us to float up the new value during the **enqueue** operation. Start by comparing the value in the root to the values of its two children. If the root's value is smaller than the values of either child, swap the value in the root with that of the larger child:



This step fixes the heap ordering property for the root node, but at the expense of possibly tweaking the subtree of the child we switched with. The subtree is now another heap where only the root node is out of order so we can recursively apply the same re-ordering on the subtree to fix it up and so-on down through its subtrees.



You stop trickling downwards when the value sinks to a level such that it is greater than both of its children or it has no children at all. This recursive action of moving the out-of-place root value down to its proper place is often called *heapify*-ing.

Your second job is to implement the priority queue as a heap array using the strategy shown above. This array should start small and grow dynamically as needed.

### General Implementation Notes

*Manage your own raw memory.* It's tempting to just use a `Vector<int>` to manage the array of elements. But using the vector introduces an extra layer of code in between your `PQueue` and the memory that actually store the elements, and in practice, a core container class like the `PQueue` would be implemented without that extra layer. Make it a point to implement your `PQueue` in terms of raw, dynamically allocation arrays of `ints`.

*Freeing memory.* You are responsible for freeing heap-allocated memory. Your implementation should not orphan any memory during its operations and the destructor should free all of the internal memory for the object. We recommend getting the entire program working without deleting anything, and then go back and carefully add freeing.

*Think before you code.* The amount of code necessary to complete the implementation work is not large, but you will find it requires a bit of thinking getting it to work correctly. It will help to sketch things on paper and work through the boundary cases carefully before you write any code.

*Test thoroughly.* I know we already said this, but it never hurts to repeat it a few times. You don't want to be surprised when our grading process finds a bunch of lurking problems that you didn't discover because of inadequate testing. The code you write has some complex interactions and it is essential that you take time to identify and test all the various cases.

### Accessing files

On the class web site, there are two starter projects: one for XCode and a second for Visual Studio. Each project contains these files:

<b>main.cpp</b>	Main module
<b>performance.h/cpp</b>	Module with performance time trial functions
<b>pqueuetest.h/.cpp</b>	Module with simple <b>PQueue</b> test functions
<b>pqueue.h</b>	Interface file for the <b>PQueue</b> class
<b>pqarray.cpp</b>	Unsorted Vector implementation of the <b>PQueue</b> class
<b>pqlist.cpp</b>	Singly-linked list implementation of the <b>PQueue</b> class

To get started, create your own starter project and add the client files as well as the desired **PQueue** implementation file that you need for the project.

### Deliverables

As always, you are to submit both a printed version of your code in lecture, as well as an electronic version via the submitter. You should turn in four files of code: they should be called **pqchunk.h/cpp** and **pqheap.h/.cpp**, and should contain your implementation of the **pqueue**. With your printed version of your code, be sure to include the completed worksheet from the next page and the answers to the thought questions that follow the worksheet. Please firmly staple all the pages and mark it clearly with your name and your section leader's name.

Summarize your implementation results in this worksheet. Fast operations may not even register as taking any time at all given the coarse granularity of the system clock, but slower operations should register as you increase the **PQueue** size. Run the performance trial on 4 different sizes—try something like  $N = 10000$ , so you record times for  $10000 (=N)$ ,  $20000 (=2N)$ ,  $40000 (=4N)$ , and  $80000 (=8N)$ . You may have to use a larger value of  $N$  to register the time on faster computers. Record the time reported for the given operations and memory used.

	Vector	Single-link	Chunklist	Heap
Memory Used N				
Memory Used 2N				
Memory Used 5N				
Enqueue N				
Enqueue 2N				
Enqueue 4N				
Enqueue 8N				
ExtractMax N				
ExtractMax 2N				
ExtractMax 4N				
ExtractMax 8N				
PQSort N (random)				
PQSort 2N				
PQSort 4N				
PQSort 8N				
PQSort N (sorted & reverse)				
PQSort 2N				
PQSort 4N				
PQSort 8N				

**Thought Questions (to be answered and handed/faxed in with assignment)**

Take the time to answer the following thought questions about your experiments. We're not looking for essays here, just a chance for you to show us that you have thought about the issues involved. A few sentences for each question would be just fine.

Identify those methods that run in linear time [i.e. in  $O(n)$  time], meaning that the execution time doubles if the **PQueue** itself is twice as large. Are there any methods that seem to run in  $O(n^1)$  time—that is, the execution time quadruples when the **PQueue** size goes up by a factor of 2? If so, which ones? Can you argue that some methods run in time that's slower than  $O(n)$  but faster than  $O(n^2)$ ? If so, what methods are they?

Repeat the performance trials for the chunk list adjusting **MaxElmsPerBlock** to smaller and larger numbers. What does this tell you about the relationship between chunk size and memory use and speed? What appears to be a fairly optimal range for the chunk size if the **PQueue** holds 2000 elements? What about 10000 or 20000?

The priority queue interface might stipulate that elements with equal priority should be processed in FIFO order. This doesn't much matter for a priority queue storing integers (where each 4 is no different than the others) but is important for the ER where three patients suffering from the same shortness of breath should be seen in order of arrival. Of the four implementations you wrote, which of them guarantee FIFO processing for equal priority elements? Explain your reasoning. For those implementations that don't, how might you adjust the code to meet this requirement?

In order to streamline the performance of a very common operation, many times you have to sacrifice the performance of some other, hopefully less frequently used operations. Would it be better to optimize **enqueue** at the expense of **extractMax** or vice versa in the priority queue?

What is the primary strength and weakness of each implementation? Which seems the most appealing if your goal was sheer speed? What if you wanted to use as little space (memory) as possible? What if you needed to get the code written and debugged in the shortest amount of time?