Assignment #6: Priority Queue

Due: Wed Mar 5th 2:15pm

For this assignment, you will be implementing a class of your own: a priority queue, which is a variation on the standard queue. The standard queue processes element in the first-in, first-out ("FIFO") manner typical of ordinary waiting lines. Queues can be handy, but a FIFO strategy isn't always what's needed. A hospital emergency room, for example, needs to schedule patients according to priority. A patient with a more critical problem will pre-empt others even if they have been waiting longer. This is a *priority queue*, where elements are prioritized relative to each other and when asked to dequeue one, it is the highest priority element in the queue that is removed. There are many practice uses for a priority queue.

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 a special kind of tree called a heap. 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, your role will be to act as the implementer only.

The interface

Ideally the priority queue would be written as a template class. However, given this is your first class implementation project, we're going to hold off on making it fully generic and instead use integer elements. This priority queue will store a collection of integers, where the integer itself is used as the priority. Larger integers are considered higher priority than smaller ones and are dequeued ahead of smaller values. Here is the basic priority queue interface:

```
class PQueue
{
  public:
    PQueue();
    ~PQueue();
    int size();
    bool isEmpty();
    void enqueue(int newElem);
    int dequeueMax();
};
```

Note the priority queue interface is quite similar to that of the ordinary queue. For the detailed specification of the behavior and usage of these functions, see the **`pqueue.h`** interface file included in the starter files. Just six member functions, how hard can that be? :-)

Implementing the priority queue

A priority queue can be implementing using a variety of data structures, each with different tradeoffs between memory required, runtime performance, complexity of code, etc. In this assignment, you will consider four different implementations. One implementation stores the queue elements in an unsorted vector. The second represents the priority queue as a sorted linked list. The third is a hybrid cross between an array and a linked list, a *chunklist*. The last represents the priority queue as specially-ordered binary tree called a *heap* (not to be confused with the heap where memory is dynamically allocated by **`new`**). The first two implementations are provided to you pre-written, the last two will be yours to write.

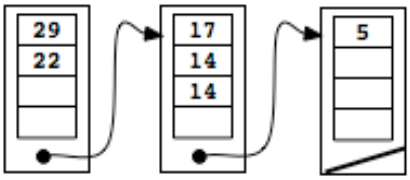
Unsorted vector implementation
The unsorted vector implementation is given to you. You do not have to write any code for this implementation, just go over and be familiar with the code provided. This implementation is layered on top of our `Vector` class. Its enqueue strategy is trivial, it simply adds the new element to the end. When it comes time to dequeue, it uses a linear search to find the maximum. You will run time trials and analyze its strengths and weaknesses.

Sorted linked list implementation
The sorted linked list implementation is also given to you. The singly-linked list is ordered by decreasing priority so as to facilitate retrieving the largest element. This arrangement makes dequeue easy, but enqueue has to search for the proper position to insert a new value. Running time trials on this implementation will show how those tradeoffs are reflected in runtime performance.

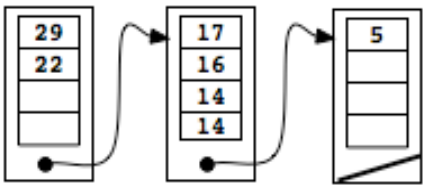
Sorted chunklist implementation
Now it's your turn! Neither the vector nor the linked list is a great performer on all operations so next you'll concoct an array/linked-list combo to see what advantages a hybrid might offer. You take advantage of the flexibility offered by a linked list but reduce some of the memory overhead and slow traversal time by making each cell not a single element, but a block of elements. The *chunklist* combines the array and linked-list concepts into a singly-linked list of blocks, where each block contains a constant-sized array capable of holding several elements.

By storing several elements in each block, you reduce the storage overhead because the pointers take up a smaller fraction of the data. 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 dequeue operation.

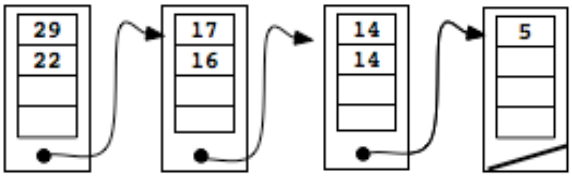
To get a better idea of how this new pqueue representation works, consider the diagram below for a pqueue with a block size of 4. Because the blocks need not be full, there are many possible representations for the same contents. A pqueue containing 29, 22, 17, 14, 14 and 5 might be divided into three blocks, as follows:



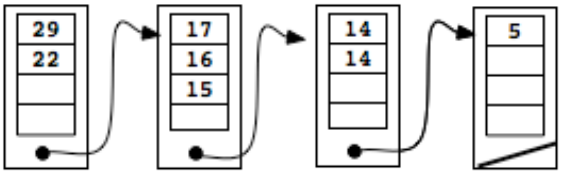
Suppose that you want to enqueue `16` and `15`. Enqueuing a `16` is a relatively simple matter because the appropriate block has only three elements, leaving room for a new one. You shift the two `14`'s toward the end of the block, but do not need to make changes to the pointers linking the blocks. The configuration after inserting the `16` therefore looks like this:



If you now try to enqueue a `15`, however, the problem becomes more complicated because the appropriate block is full. To make room for the `15`, you need to add a new block. A simple strategy is to split the current block in two, putting half of the elements into each block. After splitting (but before inserting the `15`), the pqueue looks like this:



After splitting, you can now easily insert `15`:



Your job is to implement the priority queue as a chunklist. Here are some ground rules:

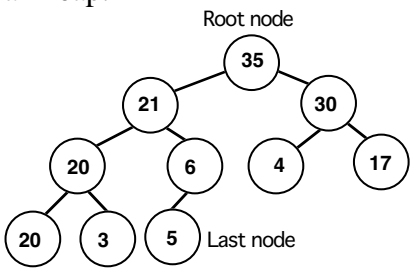
- Each list cell stores a constant-sized array. This array must be represented as a raw C++ array, not a Vector. The block size should be specified with a `MaxElemsPerBlock` constant and it should be possible to change that value to experiment with the different tradeoffs. Each block will also need to keep track of which slots are used within the block and maintain a pointer to the next block in the list.
- You must implement a sorted list of blocks but the data structure decisions beyond that are yours to make (is the list doubly-linked, does it have a dummy cell, it is circular, does you need a cursor, etc.). 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 pqueue looks like. Consider carefully how the data structures change as blocks are split.
- If you have to enqueue 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.
- There are situations where there is more than one appropriate place for an element, such as a value that could either be added to the end of one block or the beginning of the next. It is up to you to detect and decide how to best handle such a situation, but we recommend that you work toward designing a consistent, understandable strategy. In general, it will help a great deal if you try to simplify your design and avoid introducing lots of special case handling.
- Dequeuing an 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 will discuss from Chapter 13 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.

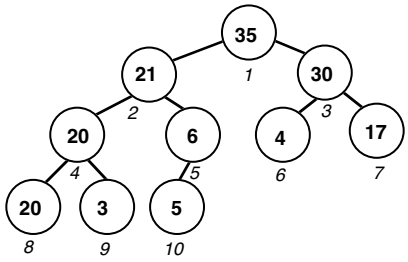
Here's a conceptual picture of a small heap:



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 the standard binary tree node definition and wire up left and right children pointers to all nodes. However, we can exploit the completeness of the tree and create a simple array representation and avoid the complexity and space overhead of pointers. Consider the nodes in the heap to be numbered level-by-level like this:



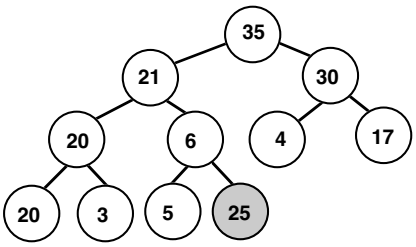
and see out this array representation corresponds to the same heap:

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

If you number the nodes starting from 1, 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. (Alternatively, you can number nodes from starting from 0 and use slightly different math to move from parent to child and back.) 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 its flattened array form.

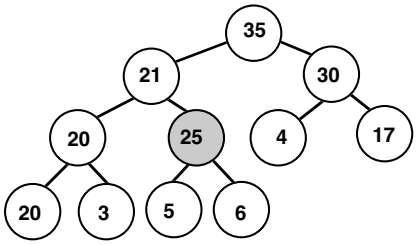
Heap insert

Inserting into a heap is done differently than its functional counterpart in a binary search tree. A new element is added to the bottom of the heap and it rises up to its proper place. For example, consider inserting 25 into the heap shown above. Add a new node with value 25 at the bottom of the heap (the next position to add is dictated by the completeness property):

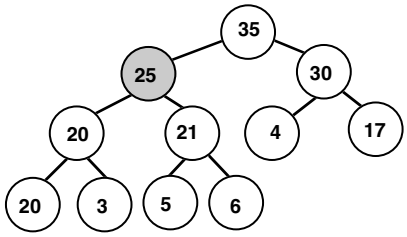


35	21	30	20	6	4	17	20	3	5	25

Compare the value in this new node with the value of its parent and, if necessary, exchange them. Since the heap is stored in array, you can move the nodes merely by swapping the values in the array. From there, 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.



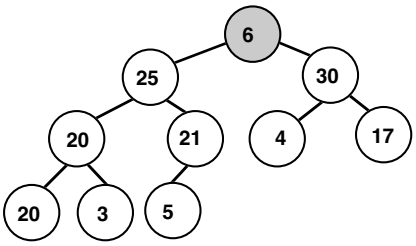
35	21	30	20	25	4	17	20	3	5	6



35	25	30	20	21	4	17	20	3	5	6

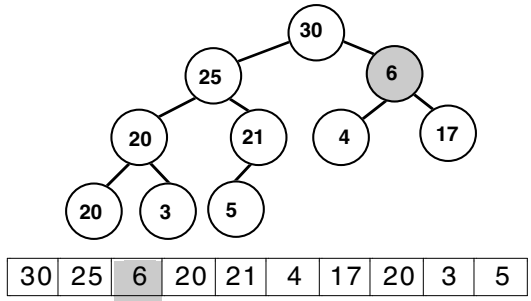
Heap remove

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 last node on the bottom level that has to go. Rather than re-arranging to fill in the gap left by the root node, leave the root node where it is, copy the *value* from the last node to the root node, and remove the last node.

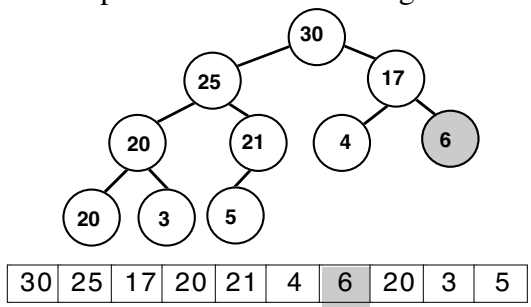


6	25	30	20	21	4	17	20	3	5

At this point, you have a complete binary tree again, whose left and right subtrees are heaps. The only problem is that the value in the root may be (and usually is) out of place. In order to restore the heap ordering property, you must trickle that value down to the right place. Use an inverse to the strategy used 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 fixes the heap ordering property for the root node, but at the expense of potentially tweaking the child that was exchanged. The child is another subheap where only the root node is out of order so apply the same re-ordering to fix it up and so-on down through as needed.



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 job is to implement the priority queue as a heap. Here are some ground rules:

- There should be no upper bound on the size of your heap, which means it must dynamically grow as needed. You can implement using a raw C++ array or layer on top of a Vector, it's your choice whether you'd like practice managing dynamic memory or accept the convenience of Vector despite its performance implications (i.e. bounds checking every access).
- Depending on how you map from node to array index (i.e. using 0 or 1-based numbering), you may have an empty slot at the beginning to take into account and/or require some adjustments in the arithmetic for computing the conversion from child indexes to parent and back.

Solution strategies
Our suggested lists of tasks to tackle for this assignment:

Task 1— Unsorted vector and sorted linked list
Read through the code in the provided vector and linked list implementations to see how they are implemented. Set up the project to use vector, run our time trial program, observe behavior and record performance in the worksheet. Swap our vector for list and repeat the experiements.

Task 2 — Sorted chunklist
Now you will get a chance to write your own implementation. Design your data structure and plot out your strategy. You might find it useful to start with our singly-linked list implementation since some of its structure is similar. The chunklist can be tricky, especially in dealing with the boundary conditions so it helps to do some thinking on paper, drawing things out and so on before diving into the code. Work on one function at a time and don't be shy about using the debugger/cout/test code to verify as you develop. You can also test your implementation using our provided client code, but you may want to write more simple tests of your own. Once you have it working correctly, run our time trial routines to test the performance of the chunklist implementation and record the results.

Task 3 — Heap
Move the skip list implementation aside and start on the heap implementation. As always, think before you code. Be sure you understand the heap structure and the transformations required for insert and remove. Implement carefully, using a one-function-at-a-time test-as-you-go strategy. When all works perfectly, run the time trials and record the results.

Task 4 — Answer summary questions
As you have seen, there are many different data structures, you could choose to represent and manipulate a priority queue, each involving tradeoffs between amount of memory required, speed of operations, difficulty of writing and maintaining the code, and so on.

For each of the implementations (the two we provided and the two additional ones you wrote), use the performance trial option available to observe and record the performance of each implementation in the worksheet. Answer the thought questions at the end of the worksheet to explore issues of time/space tradeoffs and summarize the results of your experiments.

Requirements and suggestions

- *Project logistics*. Your project should contain the .cpp files for our client test code as well as the **`pq<version>.cpp`** for the particular implementation you are actively working on. When you want to change implementations, remove any existing pqueue implementation file from the project and add the one for the version you wish to use. You must also edit the private section of the **`pqueue.h`** class interface to declare the data members and private member functions to match the implementation you are using. Our versions contain the necessary private declarations in a comment at the top of the .cpp file so you can copy and paste them into the header file.
- *Timing data*. As you've probably already realized from your earlier work with timing, the system clock data can be distorted by various artifacts, so you should run your experiments several times, throw away any outliers, and average to get a reliable result. If you've run many trials and your numbers are consistently off from the predicted big-O, it may indicate that you are mistaken about the big-O or have some lurking problems in your implementation. Consider it an opportunity for further investigation!
- *Managing 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 memory for the object. We recommend getting the entire program working without deleting anything, and then go back and carefully add deallocation. For those classes that store pointers, use the **`DISALLOW_COPYING`** macro in the private section to disallow the default memberwise copying and avoid bugs from unintended sharing.
- *Think before you code*. The amount of code necessary for the assignment is not large (about 100 lines for each class), but you will find it requires careful thinking to get correct. It helps to sketch things on paper and work through the boundary cases carefully before you write any code.
- *Debug as you develop*. Don't implement all of the member functions and try to test the entire class at once! Instead, work on one operation and test thoroughly before moving on. Our **`printDebuggingInfo`** hook can be used to dump the internal structure for you to examine. For example, after finishing writing **`enqueue`**, add test code that makes a single enqueue and prints the debugging info to see how it went. If all is well, add more enqueues with debug printing in between, and observe how the pqueue grows. Don't move on until enqueue is working correctly in all necessary cases. When writing dequeue, proceed in the same fashion.
- *Test thoroughly*. I know we already said this, but it never hurts to repeat it a few times. The code you write has some complex interactions and it is essential that you take time to identify and test all the various cases. Our test code will be helpful, but no doubt you will want to augment it with tests of your own.

Possible extensions

Never one to turn down a student who is seeking some opportunities to go further, I thought I'd share some thoughts on possible additional explorations into priority queues.

- *Add a merge operation*. One additional feature often needed when working with priority queues is the ability to merge two priority queues into one combined result. Extend your PQueue classes to support such a **`merge`** member function. A rather uninspired way to do this is to add a loop that dequeues from one and enqueues to another, but ideally, merging should take advantage of the internal implementation to operate efficiently. Add timing trials for this operation and report what efficiency you can achieve for merge operation for each of the different implementations.
- A *double-ended priority queue* supports both **`dequeueMin`** and **`dequeueMax`** operations. Extend your implementation(s) to add this facility. Which implementations can be adapted to provide to efficient access to both the min and max value?
- *Research and implement an alternative priority queue implementation*. One of the reasons I love priority queues is because that are so many possible implementation strategies, each with different strengths and weaknesses. Here are a few other data structures you might research (in alphabetical order): binomial heaps, calendar queues, d-ary heaps, Fibonacci heaps, fishspears, leftist heaps, pairing heaps, skew heaps, skip lists, and weak heaps (and there are many others!). Or might any of our CS106 classes be used to implement a priority queue? Code up one of these alternatives and add its information into your report.
- *Write some interesting client code uses a priority queue*. Priority queues can be incredibly useful in a wide variety of situations. One common use is for priority-driven algorithms, such as a heuristically-guided search or approximation algorithm that needs to efficiently access the most likely (highest priority) alternative to explore next. You will likely need to first generalize your class into template form (since storing only integers isn't likely to get you very far in terms of client code).

Accessing files http://see.stanford.edu/see/materials/icspacs106b/assignments.aspx

On the class web site, there are two folders of starter files: one for Mac Xcode and one for Visual Studio. Each folder contains these files:

main.cpp	Main program
performance.h/cpp	Module with performance time trial functions
pqueuetest.h/.cpp	Module with simple pqueue test functions
pqueue.h	Interface for the PQueue class
pqvector.cpp	Unsorted vector implementation of the PQueue class
pqlist.cpp	Singly-linked list implementation of the PQueue class
vector.h	Special version of Vector template class with additional member function to calculate memory usage (keep in project folder)

To get started, create your own starter project and add the three client files as well as the desired pqueue implementation .cpp file.

Deliverables

As always, you are to submit both a printed version of your code in lecture, as well as an electronic version via ftp. Both are due before the **beginning of lecture**. You should submit the source for the two implementations you wrote and include the completed worksheet from the next page and the answers to the thought questions. Please firmly staple all the pages and mark it clearly with your name and your section leader's name.

Today's thought on testing: "A good programmer is someone who looks both ways before crossing a one-way street." — Doug Linder

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 three different sizes—e.g. choose N = 10000 and record times for 10000, 20000, and 50000. You may have to use a larger value of N if your computer isn't as ancient as mine. Record the time reported for the given operations and memory used. Analyze your algorithms and determine the big-O analysis of the worst case performance of each operation as a function of N, the number of elements in the pqueue.

	Vector	**Single-link**	**Chunklist**	**Heap**
Memory Used N				
Memory Used 2N				
Memory Used 5N				
Enqueue N				
Enqueue 2N				
Enqueue 5N				
Enqueue Big-O				
DequeueMax N				
DequeueMax 2N				
DequeueMax 5N				
DequeueMax Big-O				
PQSort N (random)				
PQSort 2N				
PQSort 5N				
PQSort Big-O				
PQSort N (sorted & reverse)				
PSort 2N				
PQSort 5N				
PQSort Big-O				

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

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

1) Do the observed times match your big-O analysis—do functions reported to work in constant time stay fairly constant when the pqueue size grows? Do linear functions double with the pqueue size? Can you explain any big discrepancies? What effect can you see of those constant factors discarded in the big-O? Do functions that have the same complexity take the same time, i.e. if operations A and B are both O(n) do they take the same amount of time (should they?)

2) Repeat the performance trials for the chunklist changing the `MaxElemsPerBlock` constant to smaller and larger numbers. What does this tell you about the relationship between bloxk size and memory use and speed? What appears to be a fairly optimal range for the bock size if the pqueue holds 2000 elements? What about 10000 or 20000?

3) 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 the particular value of 4 being dequeued is indistinguishable from any other 4 in the pqueue) but is important for handling ER triage 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?

4) 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 `dequeueMax` or vice versa in the priority queue?

5) 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 as possible? What if you needed to get the code written and debugged in the shortest amount of time?