1 Running Time of Priority Queue Operations

In the last lecture, we gave implementations of Priority Queue operations, by storing the data in a heap. Now, we would like to analyze their running times. Since swap, array access in known positions, return etc., are all constant time, we get that peek () takes running time Θ(1) — after all, the highest-priority element is known to be at the root. For both add () and remove (), everything except trickleUp() and trickleDown() takes constant time, so we really just need to analyze the running time of those two functions.

In both cases, a constant amount of work is done in the function (trickleUp() or trickleDown()), and then, a recursive call is made with a node at the next higher or lower level. Thus, the amount of work is Θ(height(tree)). So our goal is now to figure out the height of a complete binary tree with \( n \) nodes. We will do that by first solving the inverse problem: what’s the minimum number of nodes in a complete binary tree of height \( h \)?

To make the tree have height \( h \), we must have at least one node on level \( h - 1 \). Each node at levels \( i = 0, \ldots, h - 3 \) has at least two children. Therefore, the number of nodes doubles from level to level, and we have \( 2^i \) nodes at each level \( i < h - 1 \), and at least one at level \( h - 1 \). Summing them up over all levels gives us \( 1 + \sum_{i=0}^{h-2} 2^i = 1 + (2^{h-1} - 1) = 2^h - 1 \). The sum we evaluated was a geometric series, and is one of the series that you really need to have down pat as a computer scientist.

So now, we know that a tree of \( h \) levels has at least \( 2^{h-1} \) nodes, so when \( n \geq 2^{h-1} \) (because the tree has \( h \) levels), we can solve for \( h \) and get that \( h \leq 1 + \log_2(n) = \Theta(\log n) \). So we have proved that a complete binary tree with \( n \) nodes has height \( \Theta(\log n) \).

Putting it all together, we get that both add() and remove() take time \( \Theta(\log n) \). We can now create a table of how long the different implementations take.

<table>
<thead>
<tr>
<th></th>
<th>unsorted array</th>
<th>sorted array</th>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>( \Theta(1) )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(\log n) )</td>
</tr>
<tr>
<td>peek</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>remove</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(\log n) )</td>
</tr>
</tbody>
</table>

Table 1: Comparison of Priority Queue implementations.

For other tasks we have seen in this class, the usual answer to “Which data structure is best?” was “It depends”. Here, this is not the case. A heap is clearly the best data structure to implement a priority queue. While it looks as if on individual operations, one or the other of unsorted arrays and sorted arrays will beat a heap, with a priority queue, each element is usually inserted once, removed once, and looked at once (before being removed). Thus, a sequence of \( n \) insertions, peeks, and removals will take the other data structures \( \Theta(n^2) \), while it takes a heap \( \Theta(n \log n) \), much faster.
2 C++ Standard Template Library

The ADTs we have learned about and implemented so far, and most of the ones we will learn about later, actually come pre-implemented with C++, in the Standard Template Library (STL). The STL also contains implementations of several frequently used algorithms. While many students surely have been waiting eagerly to use STL implementations instead of having to code data types from scratch, some may wonder why one would use STL. The advantage of standard libraries is that they save us some coding, and more importantly, a lot of debugging, as they presumably have been much more carefully debugged than our own code. When our own project is big enough, we’d like to avoid having to debug the basic data structures in addition to the high-level logic. The contents of STL can be roughly categorized as follows:

Containers: These are data structures; the name arises because they contain items. They can be further divided as follows:

1. Sequence containers: These are data structures in which items are accessed by the position they are stored at, like arrays. Our example from class was called List<T>.

2. Associative containers: Here, items are accessed via their “identity” or key. The user does not know which index an element is stored at — in fact, there may not be a clear index. Among the data structures we have seen so far, Bag is a natural example.

3. Adapters: these are restricted interfaces that give you specific functionality, such as Stack, Queue, or Priority Queue. They are often implemented on top of other classes, such as List<T>.

Algorithms: To use pre-implemented algorithms, you should #include<algorithm>. The algorithms all live in the namespace std. They are roughly divided as follows:

1. Search and compare: these are mostly iterator-based. They allow you to run some algorithm (compare elements to a given one, count/add all elements, etc.) on all elements of a container. To use these algorithms, you need to pass in an iterator of the container. Referring back to the discussion of the Map-Reduce paradigm we talked about in the context of iterators, these are typical functions that would contribute the “Reduce” side.

2. Sequence modification: These will affect the entire container, such as filling an array with zeros, or performing a computation on all elements. They are also iterator-based. In the context of the Map-Reduce paradigm, these algorithms would fit with the “Map” part.

3. Miscellaneous: These contain mostly sorting algorithms, and algorithms for manipulating heaps, but also moments and partitioning.

2.1 More Details on Container Classes

All containers implement at least the following functions:

bool empty();
unsigned int size();
operator= is overloaded

iterators

The size() function doesn’t tell you how many elements are in the container, but rather how many elements it can hold.

For all of the container classes described next, if you want to use them, you need to #include<container-name>, where containername is the name of the container you wish to use. You also must either use namespace std or precede each container name with std::.
2.1.1 Sequence Containers

As we mentioned above, sequence containers are similar to `List<T>` from class, in that they implement access to elements typically by index position. They allow reading, overwriting, inserting, and removing. Also, most of them implement swapping, and several other functions. There are several sequence container classes, with different tradeoffs and implementations.

`vector<T>`: This is basically our `List<T>` type, implemented using an expanding (doubling) array internally.

`array<T>`: This implements a fixed-size array, i.e., it does not grow and does not provide `insert` or `remove`. Its interface and functionality is basically the same as a standard array. (It is available only in C++11.) The main advantage over `T* a = new T[100]` is that it catches array indices out of bounds and throws exceptions, instead of leading to segfaults or other bad errors.

`list<T>`: This is a doubly-linked list. It does not allow access by index. In C++11, there is also a singly linked list as `forwardlist<T>`; it saves a bit of memory. The linked list contains some additional list-specific operations such as combining and splicing lists.

`deque<T>`: This is *not* a `deque` in the sense that it combines a `stack` with a `queue`. Rather, it is a data type similar to `vector<T>`, but implemented with non-contiguous memory blocks. (Basically, it internally keeps a lookup table which of multiple separate memory blocks to access when a particular index is requested.) Because it does not need to copy over all elements when the size grows, inserting elements is faster. On the other hand, some operations have more complex implementations and are a bit slower. If you anticipate that you will be spending a lot of time growing your arrays, consider using `deque<T>` instead of `vector<T>`.

2.1.2 Adapter Containers

The adapter containers are called this because they “adapt” the interface of another class, and are not implemented from scratch. They provide special and restricted functionality. The three adapters in STL are `stack<T>`, `queue<T>`, and `priority-queue<T>`, implementing the data types we have learned about in this class.

2.1.3 Associative Containers

In sequence containers, we access elements by their position in the container, so it matters where exactly an element is stored. In associative containers, we only know that we want to insert, remove, and look up elements, but we don’t care where or how they are stored, so long as the data structure can find them for us (by their identity or key) when we want to access them.

The simplest associative container type we have seen so far was called `Bag<T>`, though looking at it again, we now see that what it really implements is a very basic version of a mathematical *set*. It provides the following functions:

```cpp
add(T item)
remove(T item)
bool contains(T item)
```

In addition, we have alluded several times already to a data type called *dictionary*, which creates an association between a key (such as a word) and a value (such as the definition of the word). Other examples would be associating a student record with the student’s ID, or the web pages that give good answers to a search query with that query. Because the data structure creates an *association* between the key and the value, these types of structures are called *associative*. While so far, we have been calling them *dictionary*, the name used in STL and elsewhere is *map*. The functions that a dictionary/map should provide are:
add (keyType & key, recordType & record)
remove(keyType & key)
recordType & lookUp (keyType & key)

While we have written add as a function with two parameters, the way it is actually implemented in STL is with just a single parameter, which is of type pair (a struct) of the two individual types.

STL provides implementations of set (Bag) and map (Dictionary), differing in several parameters.

set<T> : implements a set that can contain at most one copy of each element, i.e., no two elements may have the same key.

multiset<T> : implements a multiset of elements, which means that it may contain multiple elements with the same key.

map : implements a map from keys to records, ensuring that there is at most one copy of each key.

multimap : implements a map from keys to records that allows the data structure to contain more than one copy of a key.

All of these data types are implemented using balanced search trees (which we will learn about in a few lectures). What this means is that you will have to provide a comparison operator for your type T if you want to use these types.

There is an alternative implementation using hash tables (which we will learn about right after balanced trees). To access those implementations instead, you put the word unordered in front of the type's name, e.g., unordered_map.

The hash table implementation is often faster, but it depends on a good implementation of a hash function, which you may have to write and provide yourself to suit your type T. On the other hand, the tree-based version is very fast at returning all entries in sorted order, since that's how they are stored internally anyway.