# Document

This document contains historical notes and lessons learned. Anything I found needed more context than the *existing,* aka turned in, code I put here instead of in the code. Comments that explain history I find to be confusing and cluttering so I limited my documentations in code to be descriptive of the final solution. I think this may contrast slightly with your preferences, but I simply found it hard to document these things in comments.

# Test Plan

The test plan here is actually split up into unit tests for the individual Node, List, PriorityString, and PriorityQueue classes. Integration tests are then performed on Priority Queue.

## Memory

The first tests verify memory usage.

1. Operations on various types of Item declarations
   1. Stack Frame
   2. Heap
2. Tests to verify handling empty things
3. Tests to handle over and underflow
4. Tests to verify construction
5. Tests to verify destructor operations

Functional Tests

1. Tests of push
2. Tests of pop
3. Tests of exceptions
4. Tests of various priorities
   1. Tests of negative priorities
5. Tests of operators
   1. Self assignment
   2. Post vs pre increment
   3. <<

Testing of things like searching and inserting

* My list implementation searches on push
* It also uses insert

# Lessons Learned

## Using the provided list as a challenge

I did this assignment twice, the first solution can actually be seen here. But I wanted to talk about how I did it. I did this as an exercise in using strictly controlled third party code. Sometimes you can’t rewrite the code at all, for instance when the code has already been through verification. But it was difficult as the provided code was hard to shoehorn and was actually incorrect in several places, which kind of defeated the purpose of the exercise.

I implemented a sorted insert routine inside the ppqueue class. The head is declared as a private Node and I needed access to the node pointer to call node insert after I had found the appropriate location. This required “breaking” the class buy returning by reference the pointer to the head with a new sub routine.

I encountered issues with inserting into an empty list with the insert methods that were provided. The lists constructor creates a List with the head node pointer set to null. The list\_head\_insert method of Node handle this by always inserting at head as we have discussed in class. The list\_insert method however accessed previous\_ptr->link() without checking for null. This would cause a null de-reference exception when inserting using this method on an empty list.

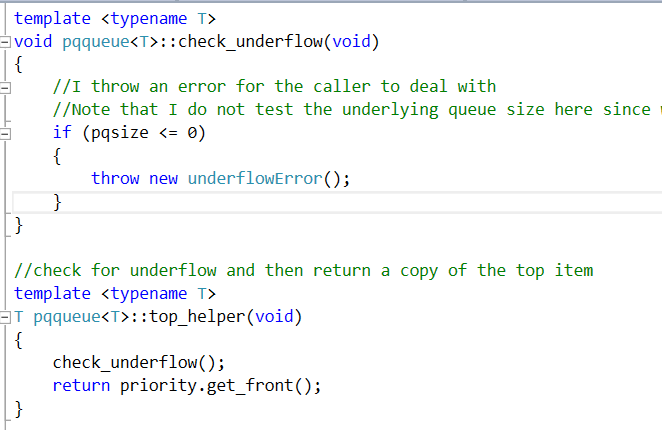
## Checking reference input arguments for null

I learned you don’t have to do this like with pointers because a reference must be bound at assignment. Now, this does not prevent the item that the reference was bound to form being deleted or corrupted etc. I made a personal decision here with regards to defensive coding to not try and verify the memory of the reference. I argue that the integrity of the item being passed to my function is the responsibility of the user as this would introduce scope creep into my list functions.

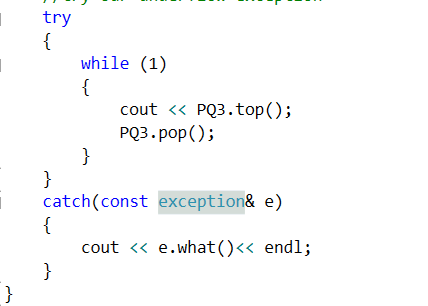
The underlying list and node functions should guard against invalid memory by checking for things like null dereferences and gracefully throwing so that the user can find their issues easily though. I did not fix the node and list routines because I wanted to use my own.

## Exception handling

C# throws error “up” until a catch block is hit. Apparently c++ only throws “up” one level. The following code still executed the get\_front despite check\_underflow throwing an error:



And having a catch block defined in the program as seen here:



## SFINAE

This is a concept “Substitution Failure Is Not An Error”. It is a method of checking to see if a Templated Type has a member function. I was going to use this to determine if a given type T had a member method getPriority() that returns an int. Currently if the type T does not contain this method, there will be a compile time error.

Unfortunately, it was fairly complex. And I focused instead on completing the assignment. Hopefully I may be able to use it when we get to heaps.

I also want to note that it is WAY easier to do this in c#. In c# you simply add what looks like an inheritance

## The power of basic\_node

This allows me to not template the iterator portions of my nodes! Cool! Imagine the fractions of kb this will save! It saves space because the iterator code will not be regenerated for each type that utilizes it.

I didn’t really get the idea from one place so much as several sources on the internet, but I implemented the code from scratch after reading. I also inspected the g++ source libraries on my computer so I guess that would be a source to cite as well.

(note that this is not in the version using your code, but the version I will submit later using my linked list)

## Iterators

I used iterators to simplify looping operations (and to do it for fun). I defined one for your list class but it is not a very polished implementation.

For my list class I had trouble figuring out exactly *where* to declare iterator to mmic the stl designs. This post helped a lot: <http://stackoverflow.com/questions/7758580/writing-your-own-stl-container/7759622#7759622>

# Errors found

There were many, many errors in the classes provided. So many that I am not sure that I have exhaustive list here. I did ensure to comment as many as I could in the code with comments so please use a diff tool to see where these changes occurred.

## Friend class declarations

The declaration of operator<< was missing its Template Parameter input.

friend ostream& operator<<(ostream& out, const pqqueue& pq);

The function declaration needs to be updated to:

template <T>

friend ostream& operator<< (ostream& out, const pqqueue<T>& pq);

I was not 100% clear on why. The solution actually came from this post <http://stackoverflow.com/questions/4014294/operator-overloading-on-class-templates>. I gathered the jest of it was related to the fact that in the friend implementation, the typename IS NOT the same T as the typename of the declaration of the function.

After further looking around I found that this declaration also works:

friend ostream& operator<< <T>(ostream& out, const pqqueue<T>& pq);

You can also add the Template Parameter to this statement if preferred, or even more interesting not specify the typename and us an open <>. Interesting.

This was also an issue with the operator+.

1>Driver.obj : error LNK2019: unresolved external symbol "class pqqueue<class PriorityString> & \_\_cdecl operator+(class pqqueue<class PriorityString> const &,class pqqueue<class PriorityString> const &)" (??H@YAAAV?$pqqueue@VPriorityString@@@@ABV0@0@Z) referenced in function \_main

(I think there are likely several ways to fix this. And I am sure that they have different implications in the amount of code generated for the template instantiation.)

# Future work

I would create a base class for priority that PriorityString inherits from that defines the getPriority method and member. This would allow more granular testing on priority form within the priority queue methods. I may also implement priority queue to operate on this class as opposed to a pure generic, and use generics in the underlying priority class.

Const\_cast indicates issues with the design. I would investigate the causes for the issues that using the cast solved and fix them, rather than “shut the compiler up”

# TODO list

Check for negative priorities

Verify reference null initialization.

The underlying list returns by value, but the pqqueue returns by reference, this leads to runtime memory exceptions