**Queues with Adapter Design Pattern**

**INTRODUCTION**

For the implementation of my project, I will be using the Class adapter pattern to adapt a doubly linked list as a queue. The problem that I will be solving is efficiently keeping track of riders in an amusement park waiting for a ride. There will be some people currently in the queue, and the current rider will scan their ticket to be added to the queue. The new rider’s ticket will be pushed onto the queue, and every 10 seconds, as people ride, 10 people will be popped off the queue. The current wait time will be updated after every ride.

**DATA STRUCTURE AND DESIGN PATTERN**

The adapter design pattern is a software design pattern that allows an interface of an existing class to be used as another interface. It can be used for compatibility by making existing classes work alongside other classes without changing the source code. The adapter design pattern can be compared with real life physical adapters. The interfaces can be thought of as receiver plugs and the adapter design pattern is the cable that makes different receivers compatible. It is needed when interfaces are incompatible but their functionality solves the problem presented. An adapter works by converting the interface of one class into an interface that is expected by a client.

There are two main types of adapter patterns. They are Object Adapter patterns and Class Adapter patterns. In Object Adapter patterns, the adapter contains an instance of the class it wraps. In Class Adapter patterns, multiple polymorphic interfaces are implementing or inheriting both the interface that is expected and the interface that is pre-existing. The Class Adapter pattern is useful when an existing class provides some or all of the services that are needed but does not use the interface that is needed.

A queue is a type of data structure in which elements are kept in order and new elements are added to the rear position. Queues are a first in first out data structure. The insertion of elements to the rear position is known as enqueue and the removal of elements from the front is know as dequeue. A queue can perform the action of a buffer when elements are stored to be processed at a later time.

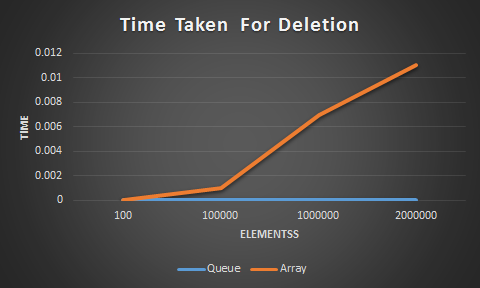
Efficient queue should be able to perform the enqueue and dequeue operations in O(1) time. Queues are often implemented as a linked list or a deque. C++’s Standard Template Library includes a queue template class that has push/pop operations. Queues should have a push\_back function which adds an element to the back of a queue and a pop\_front element which deletes the element at the front of the queue. Many queues have a pointer to keep track of the front and the end of the queue, and a member function to keep track of the size of the queue.

I implemented my queue by adapting a doubly linked list. It could have also adapted a singly linked list that keeps a pointer to the last node in addition to the first node. An array is another data structure that could be used to solve this problem, however it would be less efficient. An array would have to move all of the elements when people’s tickets are deleted from the queue. Since deletions are a major operation used in solving this problem it would be inefficient. A circular array however could be implemented as a queue and it would have constant insertions and deletions to the front and back.

**ANALYSIS**

My queue performs the enqueue and dequeue operations in O(1) time. These are the most important operations that are needed to solve the amusement park problem. This is why the queue is the ideal data structure to tackle this problem. I compared my queue to a basic array. Since we will be always be inserting to the back of the array insertion should be constant time O(1). An array would have O(n) deletion. The reason for this is during deletion the whole list will need to be shifted. Both the queue and the array have a space complexity of O(n), however an array will be more space efficient if the array is mostly full. The reason for this is the queue adapts a doubly linked list therefore there will be pointer overhead. This means in some situations if the user is very concerned about space and not concerned about an added time for deletion an array may fit their needs. However if an array is used the user would want to constantly resize the array to guarantee that the array is mostly full.

I tested my pop operation and a deletion operation for an array. As you can see my pop operation takes constant time, and the array deletion takes linear time. The graph is not completely straight since I did not plot a completely linear data set.



**IMPLEMENTATION**

For my queue implementation I adapted a doubly linked list to perform as a queue. I use a total of four classes in my implementation. I have a node class which has three member variables. These variables are next, previous, and value. Next and previous are node pointers that are used to transverse the linked list and value is the value of the element stored at the node. I also have an iterator class which has a member variable i which is a node pointer. The Iterator class was made to easily overload the ++,--, and \* operators for node pointers. I also have a linked linked class which has all of the functionalities that I needed for a queue. The list class has three member variables two of them are both node pointers, they are head and tail. They keep track of the beginning and the end of the list. The last member variable is sizeOf and it keeps tracks of the size of the linked list. My queue class has a member variable que which is of type list. Queue adapts all of its functionality from list. It has the functions pop, push, size, front, and end. It also contains an overload for the << operator.

The pop and push operations both occur in O(1) time.

Here is an example of my adaption of the push\_back and pop\_front functions in my queue.

void push(T const& x)

{

que.push\_back(x);

}

void pop()

{

que.pop\_front();

}

Now if I have a queue named que I can pop something from the front of the que by typing que.pop(). I can push the number 10 onto the back of the queue with que.push(10).

Here is an example of the pop\_front() function in the doubly linked list class.

void pop\_front() {

// make sure the list is not empty

assert(size()!=0);

// make a temporary pointer which points to head

if(size() == 1) {

head = nullptr;

delete head;

}

else {

Node<T>\* tmp = head;

// set head to point at the second node in the list

head = tmp->next;

// set the 'previous' pointer for the second node to null

tmp->next->previous = nullptr;

// delete the old head and decrement the size of the list

delete tmp;

}

--sizeOf;

This pop function deletes the front node in O(1) time provided we have a reference to the node stored at head. It also decrements the size of the linked list.

**CONCLUSION**

All in all I was looking for a data structure and design pattern to keep track of track of riders currently waiting for a ride. The main operations performed are insertion and deletion, which my queue does in O(1) time. My research confirmed my suspicions that a queue is the best data structure to solve this problem.

**REFERENCES**

<https://en.wikipedia.org/wiki/Queue_(abstract_data_type)>

<http://www.cplusplus.com/reference/queue/queue/>

<http://bigocheatsheet.com/>

<https://en.wikipedia.org/wiki/Adapter_pattern>