Queues

Topics:

+ Enqueue Method
+ Dequeue Method
+ Peek Method
+ IsEmpty Method

Resources:

- `Node.h` - `Queue.h` - `main.cpp` - `bicnt.cpp`

Introduction

A *queue* is a data structure that follows the principle first in first out (FIFO). That is, the objects are removed in the same order they are added to the queue. This behavior is seen in a number of virtial and real world applications especially for organizers and multiple thread programming. The methods of a queue are

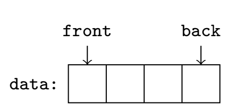
- `Enqueue()` - it adds an object to the end of the queue.
- `Dequeue()` - it removes the oldest object from the queue.
- `Peek()` - it views the oldest object in the queue if it is not empty.
- `IsEmpty()` - it determines if the queue is empty.

For this lecture, we will look at three implementations of a queue such that all methods of the queue will have a constant big-O runtime. The implementations will be container classes of the queue implemented as an array, a singly linked list and a doubly linked list. Their fields are as follow

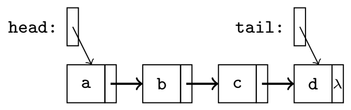
```
template<class T>            template<class T>           template<class T>
class Queue                  class Queue                 class Queue
{                            {                           {
 private:                     private:                    private:
 T* data;                     Node<T>* head;              Node<T>* head;
 ulong front;                 Node<T>* tail;              Node<T>* tail;
 ulong back;                 };                          };
 ulong capacity;
};
```

where the leftmost is the array, the middle is the singly linked list and the rightmost is the doubly linked list. Likewise, the array implementation will have a fixed size while the linked list implementations will be dynamic. Unlike a stack, a queue requires two indices (or references) of the array (or linked list) to be implemented.

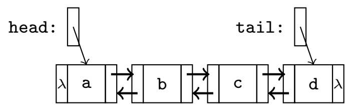
Array



Singly Linked List



Doubly Linked List



The *front* index (or *head* reference) is used to keep track of the oldest object added to the queue; hence, it is used to view and remove it. While the *back* index (or *tail* reference) is used for adding new objects to the queue. Furthermore, for the array implementation, we will make it circular to be space efficient. This also means that there will be a free space (an additional memory cell) in the array that will not be a part of the capacity of the queue; that is, the maximum capacity of the queue is one less than the capacity of the array. This free space is not fixed; its position depends on when the queue is empty.

Now, their default constructors are

```
Queue()
{
 front = 0;
 back = 0;
 capacity = 101;
 data = new T[capacity];
}
```

```
Queue()
{
 head = NULL;
 tail = NULL;
}
```

```
Queue()
{
 head = NULL;
 tail = NULL;
}
```

The array implementation will have an overloaded constructor that allows the user to assign the value of the *capacity* of the queue; however, the linked link implementations will not define one. To understand how to configure the indices (or references), we will also look at the definitions of the copy constructor which are

```
Queue(const Queue<T>& obj)
{
 front = obj.front;
 back = obj.back;
 capacity = obj.capacity;
 data = new T[capacity];

 for(int i = front;i != back;i = (i + 1) % capacity)
 {
  data[i] = obj.data[i];
 }
}
```

```
Queue(const Queue<T>& obj)
{
 head = Copy(obj.head);
 tail = head;

 while(tail->link != NULL)
 {
  tail = tail->link;
 }
}
```

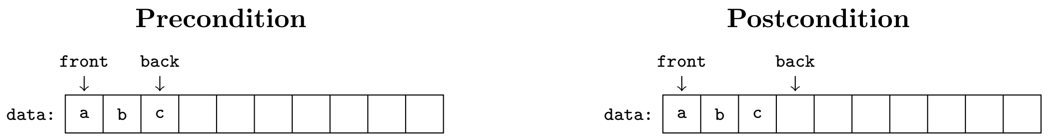
```
Queue(const Queue<T>& obj)
{
 head = Copy(obj.head);
 tail = head;

 while(tail->next != NULL)
 {
  tail = tail->next;
 }
}
```

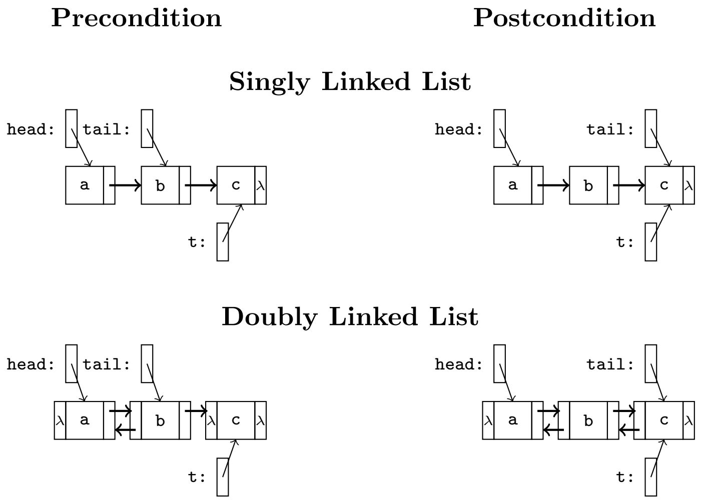
The array implementation does an in-place copy of the array; hence, the array will not necessary start from the zeroth index and work its way up to the size of the queue. However, it is possible to do so by shifting the indices. Whereas, for the linked list implementations, the tail reference needs to reference the last node of the linked list to ensure an valid and constant execution of the methods of the queue. Last, the assignment operator is similar to the copy constructor as expected and destructor will be standard implementation for its respective storage method.

Enqueue Method

The `Enqueue()` method adds to the end of the queue. Thus we need to use the *back* index (or *tail* reference). For the array, we add the new object in the current position of the *back* index, and then, increment *back* to the next available location. However, we will first check if the queue is full since the array implementation has a fixed size.



Meanwhile, for the linked lists, we will allocate a new node with the new object as its data, assign it to the *link* (or *next*) of *tail*, and then, assign *tail* its *link* (or *next*) if the list is not empty. If the list is empty, we will allocate a new node with the new object as its data, and assign it to both *head* and *tail*.



Hence, the `Enqueue()` methods will be

```
void Enqueue(const T& item)
{
 if((back + 1) % capacity != front)
 {
  data[back] = item;
  back = (back + 1) % capacity;
 }
}
```

```
void Enqueue(const T& item)
{
 if(head == NULL)
 {
  head = Create(item);
  tail = head;
 }
 else
 {
  tail->link = Create(item);
  tail = tail->tail;
 }
}
```

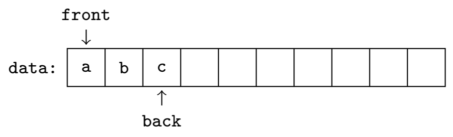
```
void Enqueue(const T& item)
{
 if(head == NULL)
 {
  head = Create(item);
  tail = head;
 }
 else
 {
  tail->next = Create(item);
  tail->next->prev = tail;
  tail = tail->next;
 }
}
```

Note that for the doubly linked list implementation, the previous link of the last node has to be assigned the previous last node so that the list is consistent.

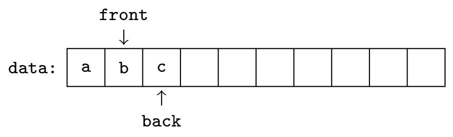
Deque Method

The `Dequeue()` method removes oldest object in the queue. Hence, the method will be using the *front* index (or *head* reference). For the array, if the queue is not empty, the *front* index is incremented to the next available position. The old location will be overridden.

Precondition



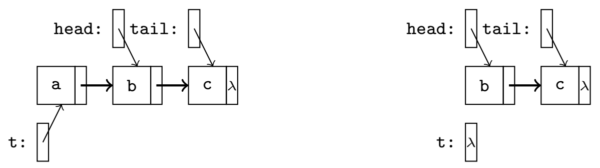
Postcondition



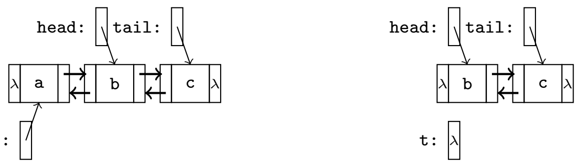
Meanwhile for the linked list, if the queue is not empty, a new node is assigned to *head*, and then, *head* will be assigned its *link* (or *next*). Afterwards, the new node will be deallocated. Moreover, if *head* is now equal to NULL, then *tail* will be assigned NULL. And for the doubly linked list, the previous link of *head* will be assigned NULL if *head* is not NULL.

Precondition **Postcondition**

Singly Linked List



Doubly Linked List



Its methods will be

```
void Dequeue()
{
 if(front != back)
 {
  front = (front + 1) % capacity;
 }
}
```

```
void Dequeue()
{
 if(head != NULL)
 {
  Node<T>* t = head;
  head = head->link;
  delete t;
  t = NULL;

  if(head == NULL)
  {
   tail = NULL;
  }
 }
}
```

```
void Dequeue()
{
 if(head != NULL)
 {
  Node<T>* t = head;
  head = head->next;
  delete t;
  t = NULL;

  if(head == NULL)
  {
   tail = NULL;
  }
  else
  {
   head->prev = NULL;
  }
 }
}
```

Peek Method

The `Peek()` method views the oldest object of the queue, which is the object that will be removed next. For this method, you just have to check if the queue is not empty. If the queue is empty, an error is thrown; otherwise, the oldest object is returned. Its methods will be

```
const T& Peek() const
{
 if(front == back)
 {
  throw "Empty Queue";
 }
 return data[front];
}
```

```
const T& Peek() const
{
 if(head == NULL)
 {
  throw "Empty Queue";
 }
 return head->data;
}
```

```
const T& Peek() const
{
 if(head == NULL)
 {
  throw "Empty Queue";
 }
 return head->data;
}
```

For the array implementation, the oldest object is the element with the index *front*. While, for the linked list implementations, the top object is the data of *head*.

IsEmpty Method

The `IsEmpty()` method determines if the queue is empty. This is accomplished with just a boolean expression for all implementations. Its methods will be

```
bool IsEmpty() const
{
 return (front == back);
}
```

```
bool IsEmpty() const
{
 return (head == NULL);
}
```

```
bool IsEmpty() const
{
 return (head == NULL);
}
```

```
bool IsFull() const
{
 return ((back + 1) % capacity == front);
}
```

The additional method `IsFull()` on the far right is for the array implementation since the queue has a fixed size. It determines if the queue is full.