CSCI-2270-Data Structures
Instructor-Hoenigman/Jacobson
Recitation 6

In this recitation, you will be learning the basic manipulations of doubly linked lists, such as, building a doubly linked list and deleting nodes from it. You will also work on implementing a queue using an array.

Doubly Linked Lists

```
//node implementation for doubly linked list
struct node{
        int key;
        node *next;
        node *previous;
};
```

Building Doubly Linked Lists

Very similar to the way we built singly linked lists, we can create Doubly linked lists. You will be adding another pointer to keep track of the other direction.

```
node *n0 = new node;
n0->key = 1;
n0->next = NULL;
n0->previous = NULL;
// Then create node 2, let's call this n1
```

To connect the two nodes set the next and previous pointers for both nodes

```
n0->next = n1;
n1->previous = n0;
```

Deleting Nodes from a Doubly Linked List

To delete a node from a linked list, update the pointers to bypass the node, and then free the memory associated with the node.

Deleting head node:

```
        //search for the node to delete.
        tmp = head                      // create temp pointer and set to head node
        head = head->next               // move head pointer
        head->previous = null           // set previous pointer to null
        delete tmp                      // delete the original head node
```

Deleting middle node:

```
node* node1=head->next;                          //set node1 to the head next node

##Traverse the linked to search the node we want to delete node1->key==deletevalue##

node1->previous->next=node1->next                //reset the next pointer for the previous node
node1->next->previous = node1->previous          //reset the previous pointer for the next node

delete  node1                                    //delete the node
```

Deleting tail node:

```
tmp = tail                                       //similar to head, starting at tail
tail = tail->previous
tail->next = null
delete tmp
```

Inserting Nodes in a Doubly Linked List

This is similar to the above examples. You will need to create a new node and instead of setting the pointers to node->previous or node->next, you set them to the new node you just created. The only difference between inserting a node into a singly linked list and a doubly linked list is that the previous pointer for a node needs to be set on a doubly linked list.

For inserting in the middle, find the previous node in the list using the search algorithm. The node is labeled *left.* The new node will be inserted after this previous node.

Inserting middle node:

```
left = head                                      //create a pointer to the head node

node* node1 = new node                           //create the new node
node1->next = left->next                         //update all the pointers
node1->previous = left

left->next->previous = node1
left->next = node1
```

Inserting head node:

```
node *n0 = new node;          //create the new node n0
n0->key = 1;
n0->next = head;
n0->previous = NULL;

head->previous = n0
head = n0
```

Queues[1]

A **queue** is a data structure that it stores a collection of elements and restricts which element can be accessed at any time. Queues are accessed first-in-first-out (FIFO): the first element added to the queue is the first element removed from the queue, much like the line at the grocery store.

Example 1:

"This class is a fun challenge"

Adding this sentence to the queue will look as follows:

tail

challenge
fun
a
is
class
This

head

Each word in the sentence occupies only one position in the queue. Words are added at the tail position and removed from the head position (see Example 1). The positions of the tail and head move as elements are added to and removed from the queue. When you have removed the sentence, it reads as: This class is a fun challenge

[1] Excerpt From: Rhonda Hoenigman. "Visualizing Data Structures." iBooks.

Terminology:

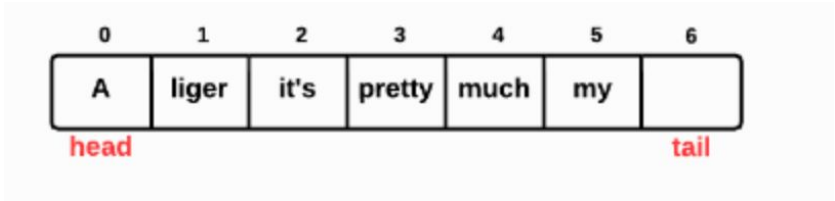
- When an element is added to a queue, it is “enqueued”. Elements are enqueued at the “tail” of the queue.

- When an element is removed from a queue, it is “dequeued”. Elements are dequeued from the “head” of the queue.

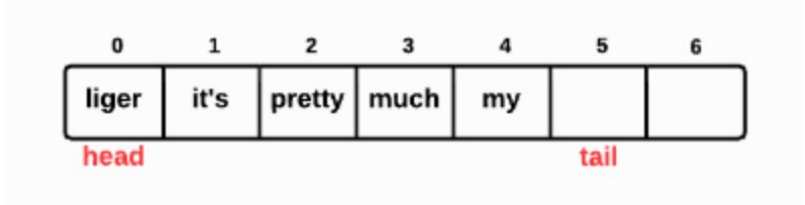
Implementation:
The data in the queue is typically stored in a data structure such as an array or a linked list. We will focus on the the array implementation.

In an array implementation of a queue, data elements are stored in an array and the head of the queue is the index where the next element will be removed and the tail of the queue is the index where the next element will be added. The elements in the array are the contents of the queue.

A dequeue() operation on this queue removes the element at the head position, which is an "A". The remaining elements are shifted to fill the space in the array. The position of the head doesn’t change, but the tail shifts by one.



After the head is dequeued, the other elements in the array are shifted by one. The position of the head doesn’t change, but the tail position shifts to the left.



Note: The simplest, but least efficient, array implementation of a queue involves shifting the elements when the head element is dequeued. Shifting the remaining elements over to fill the space is a costly array shifting algorithm.

A more efficient algorithm would be a **circular array queue**, where the queue lets the head and tail positions wrap around back to the beginning of the array as elements are enqueued and dequeued.

Features of a circular queue:

- Both the tail and the head can wrap around from the last position in the array back to the beginning. In the previous examples, when the head or tail reached Q[5], they were reset to Q[0].

- The condition where head = tail can mean that the queue is empty or full, which needs to be resolved in the enqueue() and dequeue() algorithms. Calling enqueue() when the queue is full can result in overwriting data if the algorithm doesn't check for a full queue. Calling dequeue() when the queue is empty can result in an unexpected return value if the algorithm doesn't check for a full queue. Calling dequeue() when the queue is empty can result in an unexpected return value if the algorithm doesn't check for an empty queue.

- The circular queue is more computationally efficient than a queue that uses array shifting, but it is more complicated to implement.

Enqueue operation:

With an array queue, the enqueue() operation needs to include a check for if the queue is full. There are multiple ways to check this, and the simplest approach is to keep a count of the number of elements in the queue and the queue size, and only add elements when there's room. The Queue ADT includes variables for queueSize and maxQueue. Then queueSize = maxQueue, the queue is full. The enqueue() algorithm is shown below:

Algorithm enqueue(value): Add the specified value to the queue at the tail position.

Pre-conditions
value is a valid queue value.

Post-conditions
value has been added to the queue at the tail position, queue[tail] = value.
The tail position increases by 1.

enqueue(value):

```
1.  if (!isFull())
2.      data[tail] = value
3.      queueSize++
4.      if (tail == data.end)
5.          tail = 0
6.      else
7.          tail++
8.  else
9.      print("queue full")
```

Dequeue operation:

In the dequeue() operation, there is a check for if the queue is empty. If not, the element at the head position is returned. The tail position is unchanged in the dequeue() operation. The dequeue() algorithm is shown below:

Algorithm dequeue(): Remove the queue element at the head position.

Pre-conditions
None

Post-conditions
Value at data[head] returned.
head moves by one position in the array.

dequeue():

```
1.  if (!isEmpty())
2.      value = data[head]
3.      queueSize--
4.      if (head == data.end)
5.          head = 0
6.      else
7.          head++
8.  else
9.      print("queue empty")
10. return value
```

In class work:

For your in class work, you will show the state of the queue for the following set of dequeue() and enqueue() operations in a **circular array queue**. The initial state of the queue is shown below, where the head is Q[0] and the tail is Q[5].

Array Implementation:
Q[0] = 5
Q[1] = 10
Q[2] = 6
Q[3] = 4
Q[4] = 5
Where Q[5] and Q[6] remain empty

Operations:
• dequeue()
• dequeue()

• dequeue()
• dequeue()
• dequeue()
• enqueue(8)
• enqueue(1)
• enqueue(2)
• enqueue(5)
• enqueue(7)
• enqueue(13)
• dequeue()
• dequeue()

Show the work to your TA for participation credit.

Recitation 5 Programming Assignment

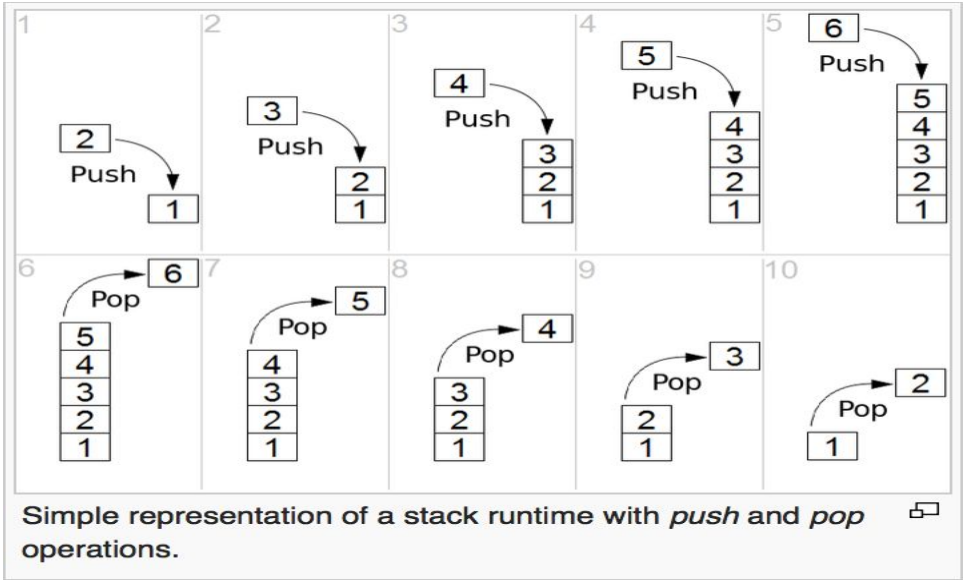
There is a link on Moodle to a Recitation 6 Programming exercise. In the quiz, you are asked to write a C++ function to write a function to perform the enqueue operation on a circular queue. You have until Sunday at 5pm to complete the exercise.

For your own reading about Stacks:

Stacks

A stack is an abstract data structure that stores a collection of elements and restricts which element can be accessed at any time. Stacks work on a *last in, first out principle* (LIFO): the last element added to the stack is the first item removed from the stack, much like a stack of cafeteria plates. Elements are added to the top of the stack, and the element on the top is the only element that can be removed. Since they are stacked on top of one another, you can't access one further down the stack until the others on top of it are removed.

Visual example of stack implementation:[2]



Simple representation of a stack runtime with *push* and *pop* operations.

Definitions:

When an element is added to a stack, it is "**pushed**" onto the stack.

When an element is removed from a stack, it is **"popped"** off the stack.

The stack ADT (abstract data type)

The stack ADT includes a variable that tracks the *top* of the stack, the stack *data*, and *methods* to manipulate the stack by adding and removing elements. *Stack data* is typically stored in a data structure such as an array or a linked list. The terminology for interacting with the stack is the same regardless of the data structure used, but the implementation details vary. The stack ADT shown in the next example is intentionally generic due to the differences in an array or linked list implementation.

---

[2] Excerpt from: https://en.wikipedia.org/wiki/Stack (accessed 2/13/2016)

Stack:

```
1.    private:
2.          top          //top of the stack
3.          data         //stack data (in array or list)
4.          maxSize
5.    public:
6.          Init()
7.          push(value)
8.          pop()
9.          isFull()
10.         isEmpty()
```

Pushing and popping stack elements

Array implementation of a stack

In an array implementation of a stack, data elements are stored in an array and the *top of the stack refers to the index* where the next element will be added. The elements, data[0... top-1] are the contents of the stack.

- When top = 0, the stack is empty.
- When top = maxSize, the stack is full. (maxSize is the size of the array)
- When top > maxSize, the condition is called stack overflow. Yes, it's called stack overflow.

Push an element onto an array stack

The algorithm to push an element onto a stack implemented with an array is shown in Algorithm below.

Pre-conditions:

value is a valid input value. A method, isFull() exists to check for if the stack is full.

Post-conditions:

The value is added to the stack and the top index is incremented by 1.

Algorithm push(value)

```
1.  if(!isFull())
2.      data[top] = value
3.      top = top + 1
```

In this algorithm, data is the stack data structure and value is the element to add to the stack. The parameter top is initialized to 0 when the stack is initially created. On Line 1, the conditional to check for a full stack calls the isFull() method in the ADT, which checks if top = maxSize.

Pop an element from an array stack

The algorithm to remove the top element from an array stack is shown in below.

Pre-conditions:
None

Post-conditions:
Element at the top of the stack is returned and top decremented by 1.

Algorithm pop()

```
1.      if top == 0
2.              print("underflow error")
3.      else
4.              top = top - 1
5.      return data[top]
```