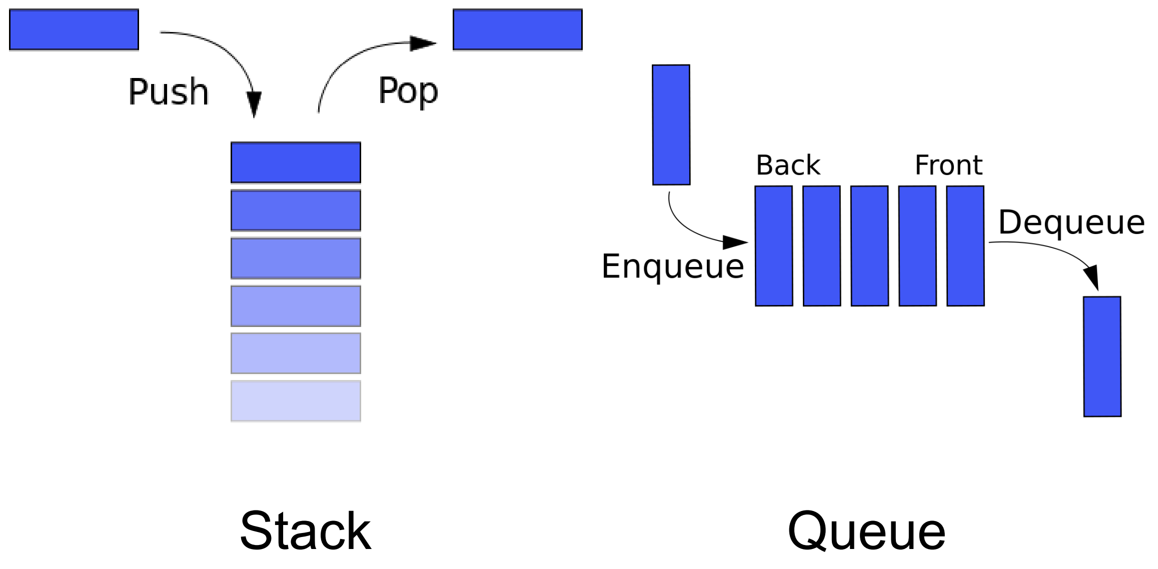
# CSCI 2270 – Data Structures

Recitation 3, Summer 2020

## Stacks and Queues

### Objectives

1. Stacks Basics
2. Implementation of stacks using arrays
3. Implementation of stacks using linked lists
4. Queues Basics
5. Implementation of queues using arrays
6. Exercise



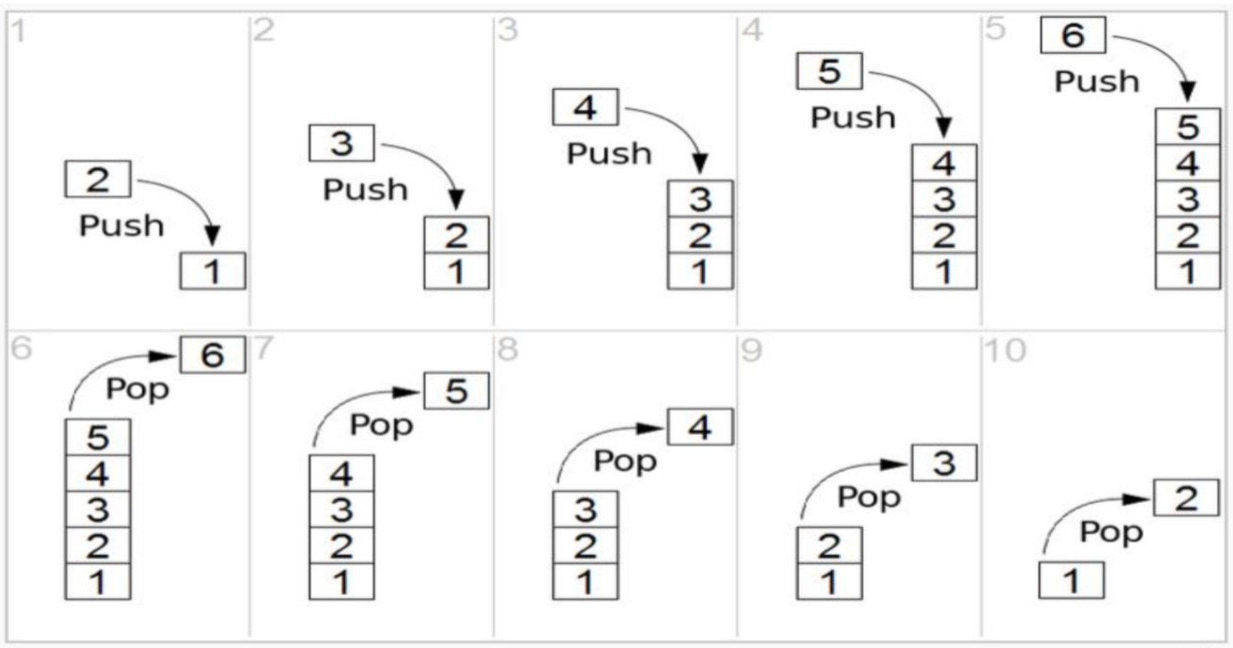
# CSCI 2270 – Data Structures

Recitation 3, Summer 2020

Stacks and Queues

## 1. 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.



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

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.

### 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.

# CSCI 2270 – Data Structures

Recitation 3, Summer 2020

Stacks and Queues

## 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.

```
class Stack
{
     private:
          top  //top of the stack
          data //stack data (in array or list)
     public:
          init()
          push(value)
          pop()
          isFull()
          isEmpty()
}
```

## 2. 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.

Pros: No pointers involved
Cons: Not dynamic. Does not grow or shrink depending on the need during runtime.

• 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.

# CSCI 2270 – Data Structures

Recitation 3, Summer 2020

# Stacks and Queues

## Push an element into 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.

```
void push(int value)
{
  if(!isFull())
  {
     data[top] = value;
     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. The condition 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 below.

### Pre-conditions

None

### Post-conditions

Element at the top of the stack is returned and top decremented by 1.

```
void pop()
{
      if(top == 0)
      {
            print("underflow error");
      }
      else
      {
            top = top - 1;
      }
      return data[top]
}
```

## 3. Linked List implementation of a stack

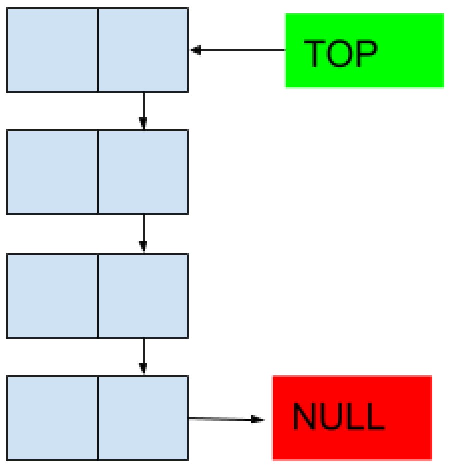
Pros: Can grow and shrink according to the needs at runtime.
Cons: Requires extra memory due to involvement of pointers.

The stack implementation using linked lists works for variable size of data.There is no need to fix the size of the stack in the beginning. Every new element is inserted at the head of the list.Therefore every new element is pointed by the **top** pointer. Whenever we want to remove an element, simply remove the node pointed by the **top** pointer, and moving the top to next node in the list.

### Steps to insert a node in a stack using linked lists.

1. Create a new node with a given value.
2. Check whether stack is empty (top == NULL).
3. If it is empty, point newNode’s next pointer to NULL.
4. If not empty, newNode’s next pointer should point to the value of top.

5. Now update top to point to the new node.



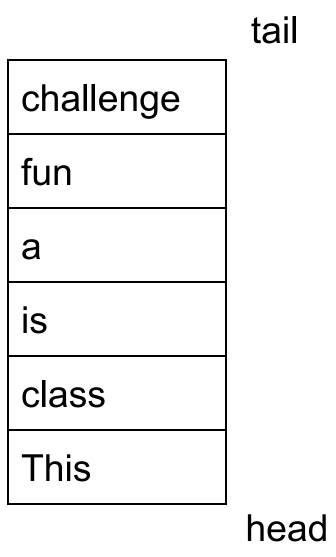
## Steps to pop a node from the stack using linked lists

1. Check whether the stack is empty. (top == NULL)
2. If empty, we can't pop any element. So, display an error message.
3. If not empty, create a new pointer temp and point it to top.
4. Update top to the next node. (top = top->next)
5. Delete temp.

## 4. Queues

A queue is a data structure that 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:



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.

**Terminology:**

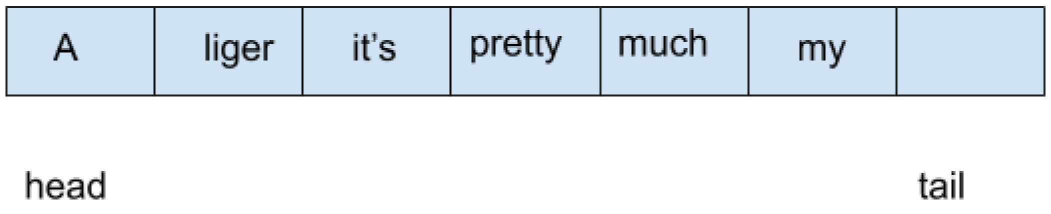
- When an element is added to a queue, it is "enqueued". Elements are enqueued at the "tail" of the queue.
- When an element is removed from a queue, it is "dequeued". Elements are dequeued from the "head" of the queue.

## 5. Array implementation of queues

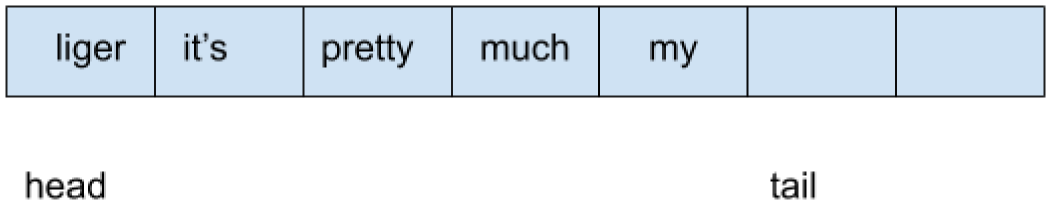
The data in the queue is typically stored in a data structure such as an array or a linked list.

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.

**Abstract Data Type for Queues**

```
class Queue
{
      private:
            head
            tail
            queueSize
            capacity
      public:
            init()
            enqueue(value)
            dequeue()
            isFull()
            isEmpty()
}
```

## Circular Arrays for Queues?

First, let us see whether we can use simple arrays for implementing queues as we have done for stacks. We know that the insertions are performed at one end and deletions are performed at the other. Once we start removing elements from the queue, the initial elements of the array will start getting wasted. To reuse that space we use circular queue. Or else we will need an array of infinite size as we perform multiple operations.

## 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 capacity. If queueSize = capacity, the queue is full. The enqueue() algorithm is shown below:

### 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.

```
void enqueue(value)
{
      if (!isFull())
      {
            queueSize++;
            data[tail] = value;
            tail = (tail+1)%capacity;
      }
      else
      {
            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:

### Pre Conditions

None

### Post Conditions

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

```
int dequeue()
{
      if (!isEmpty())
      {
            queueSize--;
            value = data[head];
            head = (head+1)%capacity;
            return value;
      }
      else
      {
            print("queue empty");
      }
 }
```

# CSCI 2270 – Data Structures

Recitation 3, Summer 2020

Stacks and Queues

## Exercise

Download the recitation exercises from moodle. This file consists of header files and function implementations of both Stacks and Queues. Your task is to complete the following function/functions:

1. **Complete the enqueue and dequeue operations of a queue. After completing these functions in QueueLL.cpp, run the DriverQueue.cpp to check if they are working correctly** (Silver problem - Mandatory )
2. Check if parentheses is balanced in an input string in Driver.cpp (Gold problem)