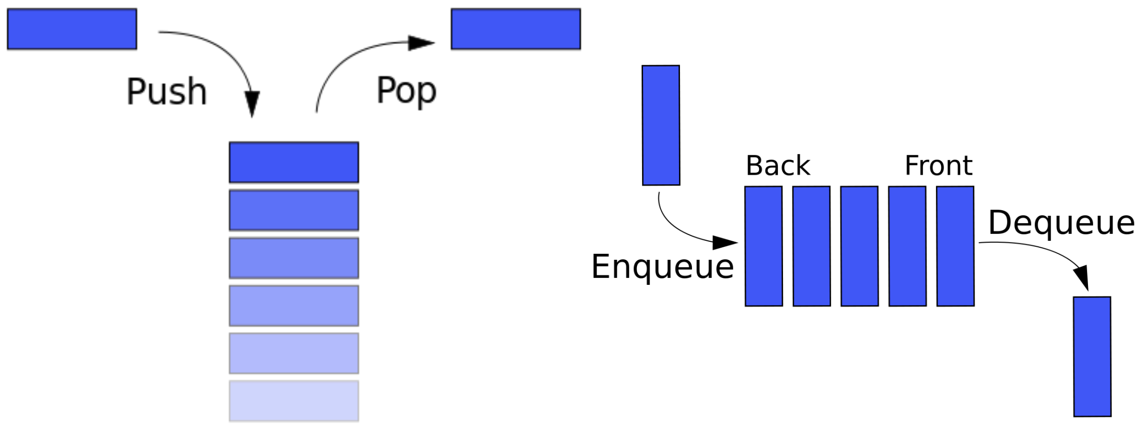
CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

Objectives

1. Stack - Basics
2. Implementation of a Stack using Arrays.
3. Implementation of a Stack using Linked Lists.
4. Queue - Basics
5. Implementation of a Queue using Arrays.
6. Exercise



Stack Queue

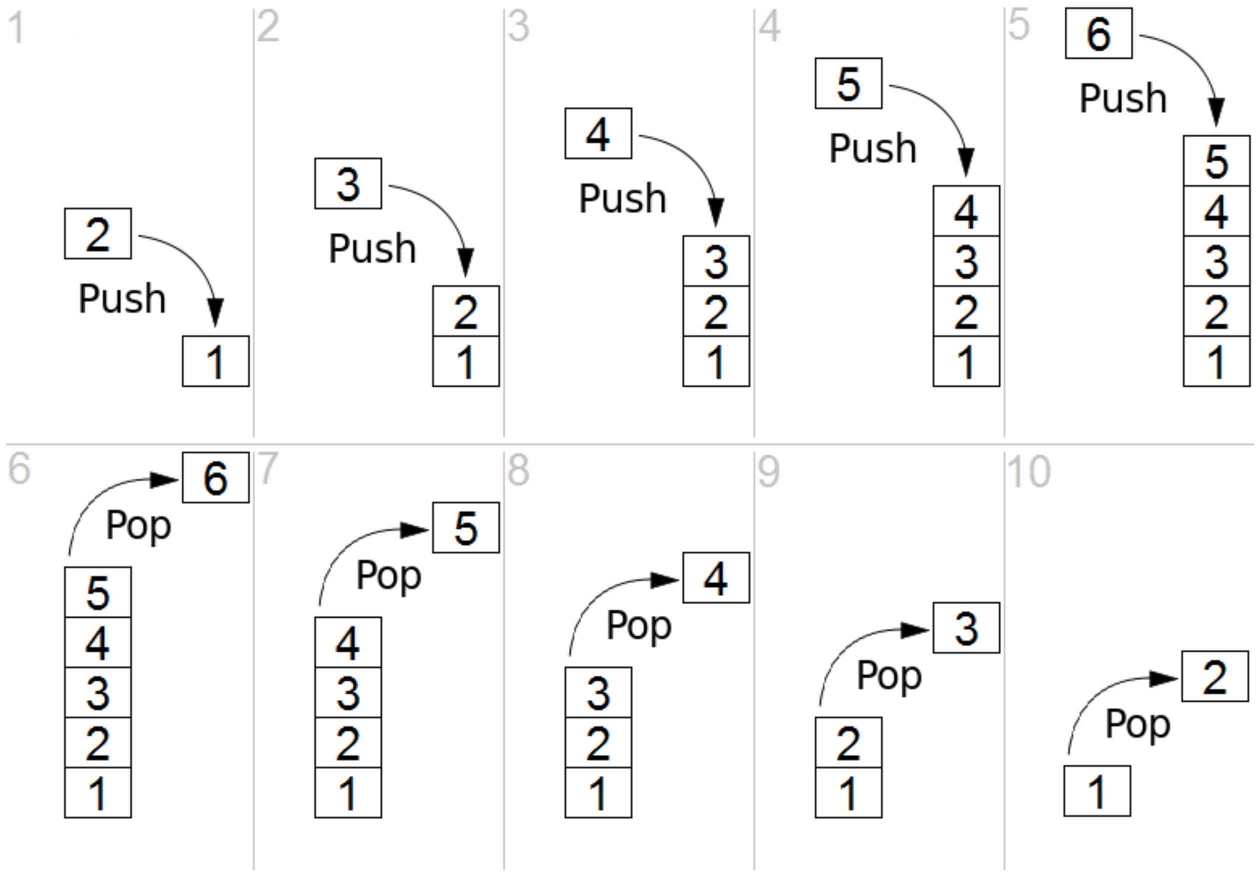
CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

1. Stacks

Stack is a one type of a Linear Data Structure. A Data Structure is said to be linear if its elements form a sequence. The abstract data structure (ADT) Stack stores a collection of elements. It restricts the access of elements based on the order in which the elements are added to it. It works on a **last in, first out principle (LIFO)**: the last element added to the stack is the first item to be removed from the stack, much like a stack of cafeteria plates.



As you can see from the figure above, elements added gets “push”-ed to the top of the Stack. And, Only, the element at the top of the stack can be “pop”-ped. Much like a stack of plates, one cannot remove any plate from the middle of the stack without removing the plates that are above it.

CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

Stack Operations

Push – Element is pushed to the top of the Stack.
Pop – Element is removed from the top of the Stack.

The Stack ADT (Abstract Data Type)

As a part of the ADT, a Stack has following variables -

Variables

- top – a variable which fetches the topmost element of the Stack.
- data – An Array or a Linked-List that holds the element(s) of the Stack.

Note: - The terminology for interacting with the stack is the same regardless of its underlying implementation - Linked List or Array.

```
class Stack {
    private:
        top;
        data;
    public:
        void init(); // Initializes the private member variables
        void push(int value); // pushes elements to the top of the Stack
        int pop(); // pops the topmost element from the Stack
        bool isFull(); // Checks if the Stack is full
        bool isEmpty(); // Checks if the Stack is empty
};
```

2. Stack – Array Implementation

In an array implementation of a Stack, data elements are stored in an array with its ***top*** variable referring to the next empty index. The elements, `data[0... top-1]` are the contents of the stack.

Pros: No pointers are involved.
Cons: Not dynamic. Does not grow or shrink at runtime.

• When `top = 0`, the stack is empty.

CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

- When top = maxSize, the stack is full. (maxSize is the size of the array)
- When top > maxSize, the condition is called stack overflow.

Push

The algorithm to push an element onto a Stack (Array implementation) is shown below.

Pre-conditions

We check to ensure the value is a valid input (of the same datatype). A method, isFull() is called 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 Stack::push(int value)
{
  if(!isFull())
  {
    data[top] = value;
    top = top + 1;
   }
}
```

Pop

The algorithm to remove the top element from a Stack is shown below.

Pre-conditions

Check if the array is empty by calling isEmpty() and print the underflow error if it returns True.

Post-conditions

If isEmpty() returns False, the element at the top of the Stack is returned and top is decremented by 1.

```
int Stack::pop()
{
    if(isEmpty())
    {
        print("underflow error");
    }
    else
    {
        top = top - 1;
    }
    return data[top]
}
```

3. Stack – Linked List implementation

Pros: Can grow and shrink at runtime.
Cons: Requires extra memory as every element now stores pointers.

Implementation of Stack using Linked List works when we do not know the size of the data at compile time. Every new element is inserted at the Head of the Linked List. The new element is always pointed by the top pointer. For the pop operation, we remove the node pointed by the top (head), free its memory and make the top point to the next node of the Linked List.

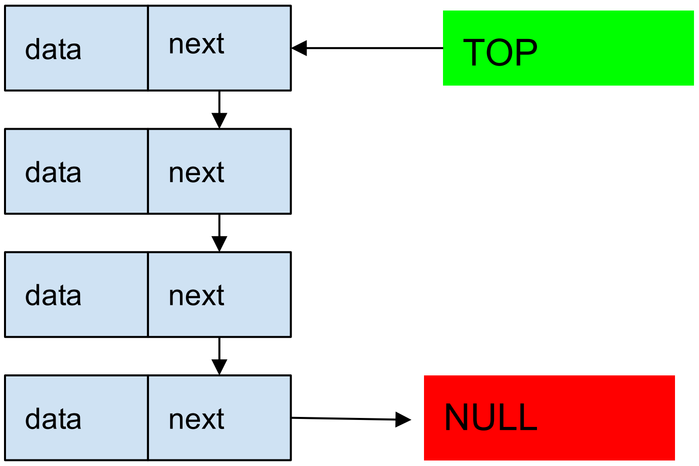
CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

Stack Insert algorithm

1. Create a new Node with a given value.
2. Check whether stack is isEmpty (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 address of top.
5. Now update top to point to the address of new Node.



Stack pop algorithm

1. Check whether the Stack isEmpty. (top == NULL)
2. Display an error message if the stack is empty as we cannot pop from an empty Linked List.
3. If the Stack is not empty, create a new pointer del and point it to the current address of top.
4. Update top to point to the next node's address. (top = top->next)
5. Delete the del pointer.

CSCI 2270– Data Structures

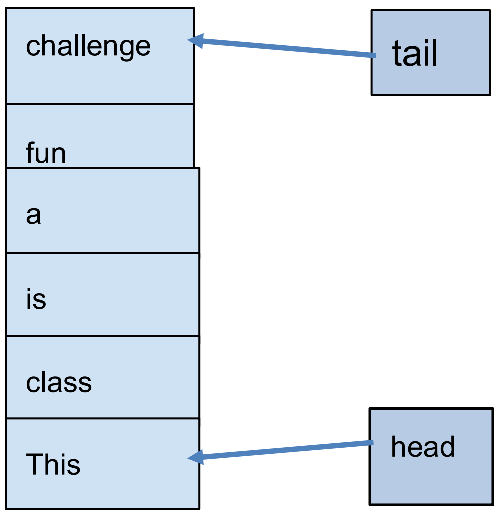
Recitation 6

Stacks and Queues

4. Queues

A Queue is another linear data structure that stores a collection of elements and restricts accessing all but the boundary elements. Queues are accessed first-in-first-out (FIFO): the first element added to the queue is the first element removed from it, similar to the line at a grocery store.

Example 1:
"This class is a fun challenge"
Adding this sentence to the queue looks 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. The positions of the tail and head move as elements are added to and removed from the queue. When all the words are removed from the Queue, your string reads: This class is a fun challenge

CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

Terminology:

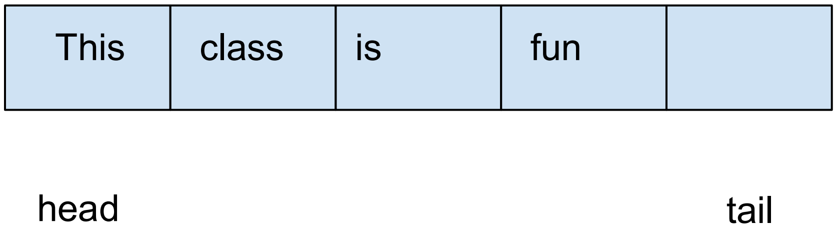
- **Enqueue** - When an element is added to a queue. Elements are enqueued at the tail end of the queue.
- **Dequeue** - When an element is removed from a queue. Elements are dequeued from the head of the queue.

Queue can be implemented using arrays or Linked Lists. Here, we will cover the implementation of Queue using Arrays.

5. Queue – Array implementation

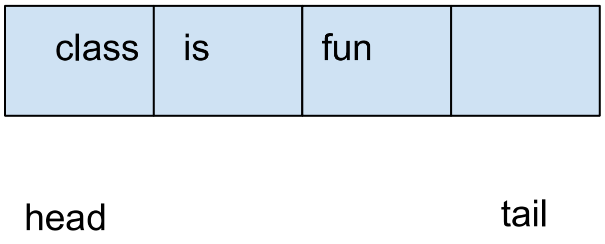
In its array implementation of a Queue, the elements are stored in an array and the head of the queue is the index of the first element to be removed, and the tail of the queue is the index where the next element to be added. The elements of the array are the contents of the Queue.

A dequeue() operation on this queue removes the element at the head ("A" here). ***The remaining elements are shifted left to fill the space in the array***. Thus, the position of the head does not change, but the tail shifts by one.



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

Queue ADT



CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

```
class Queue {
    private:
        head;
        tail;
        queueSize;
        capacity;
    public:
        void init(); // Initialize the private member variables
        void enqueue(value); // adds the element to the head of the Queue
        int dequeue(); // removes the element from the head of the Queue
        bool isFull(); // Checks if the Queue is Full
        bool isEmpty(); // Checks if the Queue is Empty
};
```

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 operation.

Circular Arrays for Queues?

We can use Circular Arrays To avoid the costly operation of shifting the elements 1 position to its left on dequeue operation. As enqueue is done on one end of the Array and dequeue is done at the other end of the Array, we can use modulo operations to increment the position of the tail/head index of enqueue/dequeue respectively. The modulo operations ensures that an index on reaching the queueSize index of the array resets to 0.

Enqueue

Using circular arrays, the enqueue operation cannot use the index of the tail to check if a queue is full. Instead, it must maintain an additional variable called queueSize that increases/decreases on enqueue/dequeue operations. The Queue ADT has a variables queueSize and capacity. Thus, if queueSize = capacity, the queue is full. The enqueue() operation is shown below:

Pre-conditions

The element to be inserted is of the same datatype as the element of the Queue.

CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

Post Conditions

The element value is added to the queue in the index tail, queue[tail] = value, and the index is incremented by 1.

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

Dequeue operation

We use the queueSize variable to check if the Queue is empty or not. We return the element of the Queue indexed at head and increment its position by 1 (ensuring modulo operation). The algorithm is shown below.

Pre-conditions

None

Post-conditions

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

CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

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

Quizzes

1. Any practical examples of stacks that we use in our daily life?
2. Any practical examples of queues that we use in our daily life?
3. What's the purpose of using a circular array for queue implementation?

Exercise

Download the Recitation 6 zipped file from Canvas. The exercise folder 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 DriverStack.cpp (Gold problem)

CSCI 2270– Data Structures

Recitation 6

Stacks and Queues

Submission: Once you've completed the exercises, zip all the files up and submit on the Canvas link. Name the submission in the following format: recitation6_firstname_lastname.zip