Data Structures Lab
Session 5

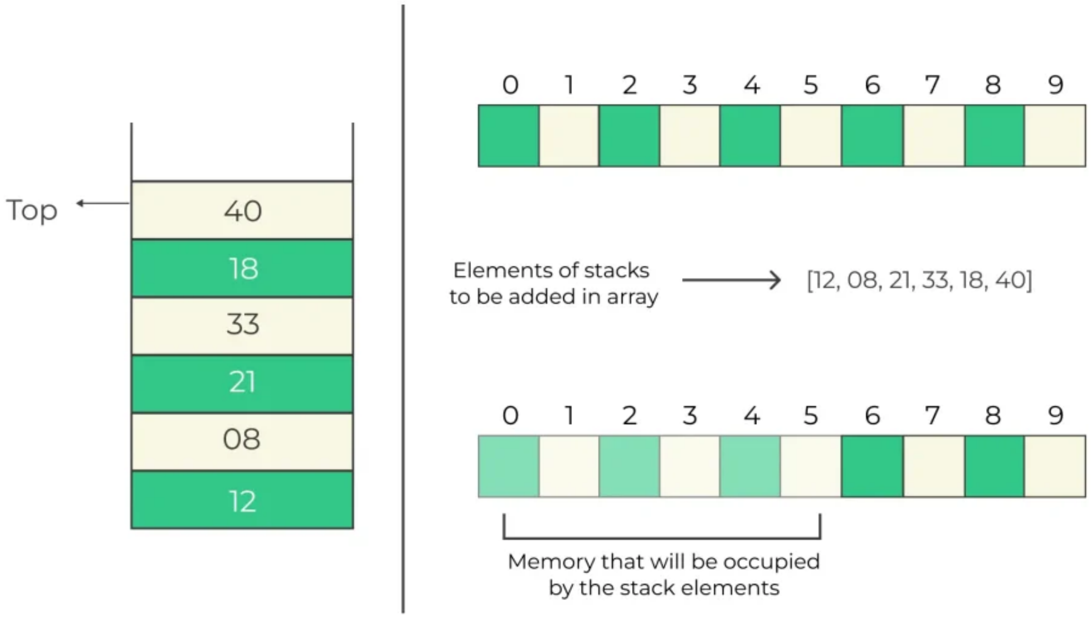
Course: Data Structures (CL1002) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif

Note:
- Lab manual cover following below recursion topics
  {Stack, Base Condition, Direct and Indirect Recursion, Tailed Recursion, Nested Recursion}
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.
- Recursion Visualization Tool (https://visualgo.net/en/recursion)

Stack with Array

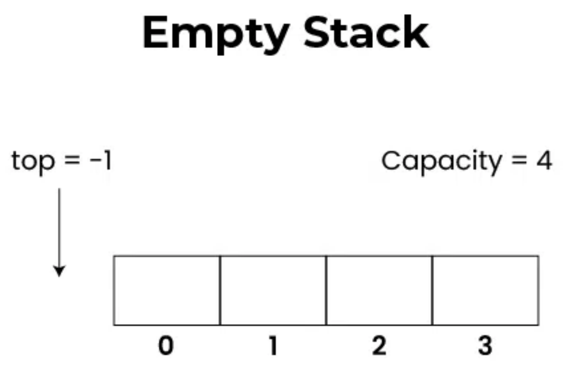
A Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last. Mainly the following three basic operations are performed in the stack:

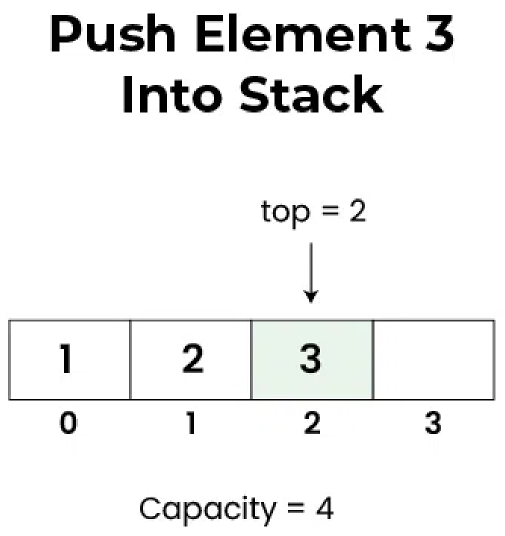
- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.
- **isFull:** Returns true if stack is full, else false.

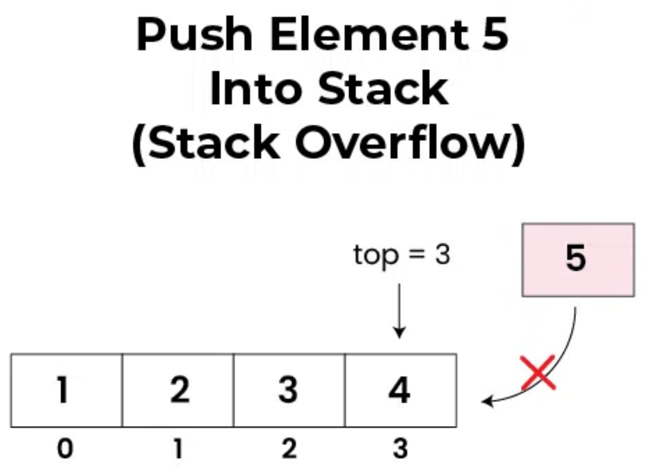

Top
40
18
33
21
08
12
0 1 2 3 4 5 6 7 8 9
Elements of stacks to be added in array
[12, 08, 21, 33, 18, 40]
0 1 2 3 4 5 6 7 8 9
Memory that will be occupied by the stack elements

<u>Sample Code of Stack in Array:</u>

```
const int MAX = 100
class Stack
{
    public:
        int top
        int a[MAX]
        Stack()
        {
            top = -1
        }
        bool push(int x)
        {
            if (isFull())
            {
                print "Stack Overflow"
                return false
            }
            else
            {
                top = top + 1
                a[top] = x
                print x
                return true
            }}
        int pop()
        {
            if (isEmpty())
            {
                print "Stack Underflow"
                return -1
            }
            else
            {
                int poppedElement = a[top]
                top = top - 1
                return poppedElement
            }}
        int peek()
        {
            if (isEmpty())
            {
                print "Stack is Empty"
                return -1
            }
            else
            {
                return a[top]
```







```
    }}
bool isEmpty()
{
    return (top < 0)
}
bool isFull()
{
    return (top >= MAX - 1)
}
int size()
{
    return (top + 1)
}
void display()
{
    if (isEmpty())
    {
        print "Stack is Empty"
        return
    }
    print "Stack elements: "
    for (int i = 0 to top)
    {
        print a[i]
    }
}};
```

Stack with Linked List

Stack Operations:

1. **push():** Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.
2. **pop():** Return the top element of the Stack i.e simply delete the first element from the linked list.
3. **peek():** Return the top element.
4. **display():** Print all elements in Stack.

Push Operation:

- Initialise a node
- Update the value of that node by data i.e. **node->data = data**
- Now link this node to the top of the linked list
- And update top pointer to the current node

Pop Operation:

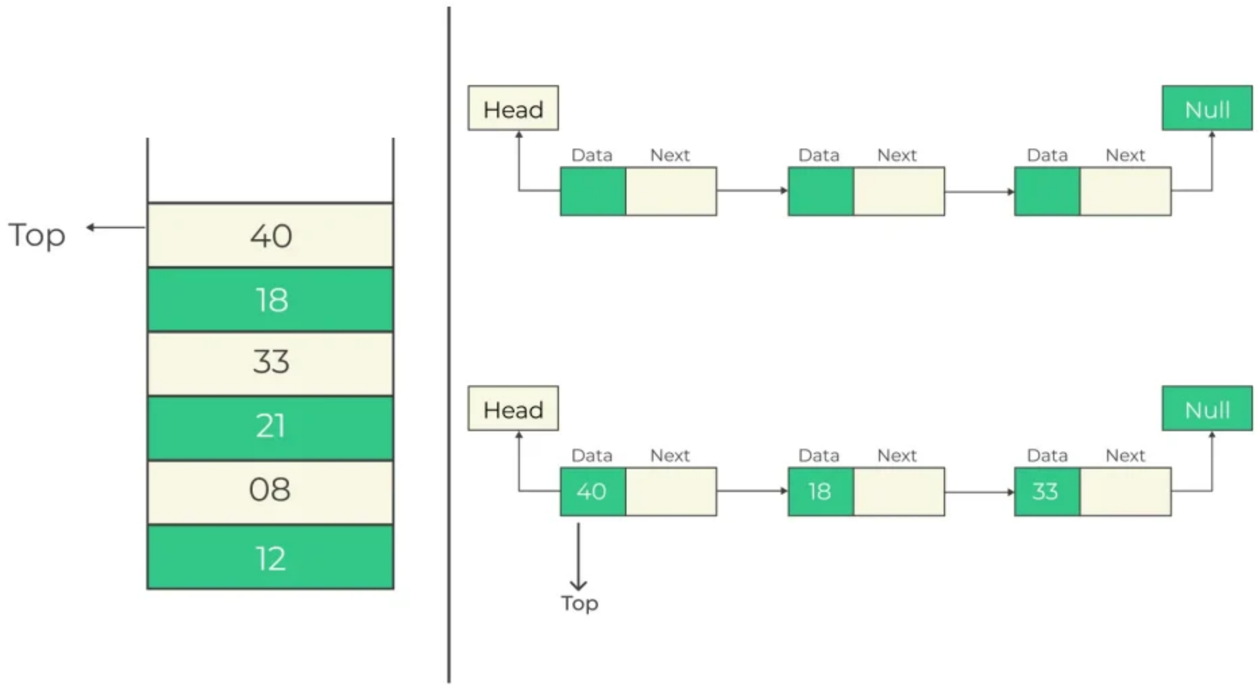
- First Check whether there is any node present in the linked list or not, if not then return
- Otherwise make pointer let say **temp** to the top node and move forward the top node by 1 step
- Now free this temp node

Peek Operation:

- Check if there is any node present or not, if not then return.
- Otherwise return the value of top node of the linked list

Display Operation:

- Take a **temp** node and initialize it with top pointer
- Now start traversing temp till it encounters NULL
- Simultaneously print the value of the temp node

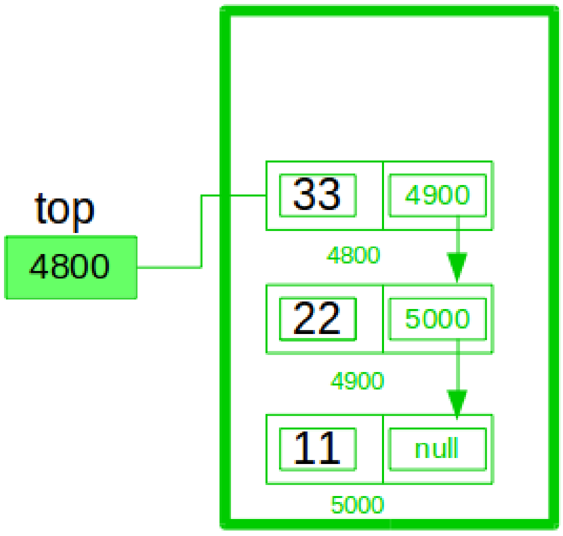


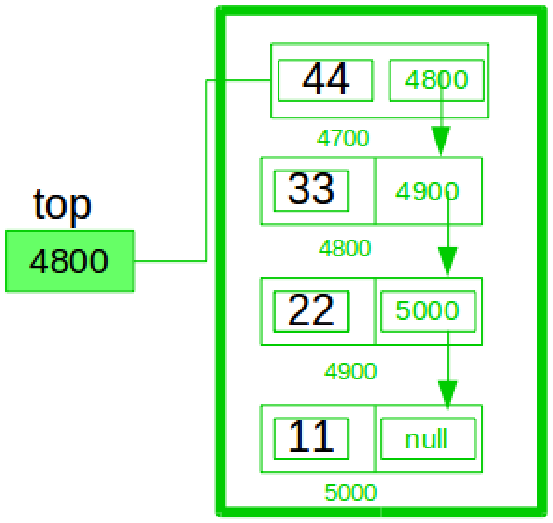
<u>Sample Code of Stack in Linked List:</u>

```
const int MAX = 100
class Node
{
    public:
        int data
        Node* next

        Node(int value)
        {
            data = value
            next = nullptr
        }
}
class Stack
{
    public:
        Node* top
        int size
        Stack()
        {
            top = nullptr
            size = 0
        }
        bool push(int data)
        {
            if (isFull())
            {
                print "Stack Overflow"
                return false
            }

            Node* newNode = new Node(data)
            newNode->next = top
            top = newNode
            size = size + 1
            print data
            return true
        }
        int pop()
        {
            if (isEmpty())
            {
                print "Stack Underflow"
                return 0
            }
            int data = top->data
            Node* temp = top
            top = top->next
            delete temp
            size = size - 1
            print data
            return data
        }
        int peek()
```





```
        {
            if (isEmpty())
            {
                print "Stack is Empty"
                return 0
            }
            return top->data
        }

        // Check if the stack is empty
        bool isEmpty()
        {
            return top == nullptr
        }

        // Check if the stack is full
        bool isFull()
        {
            return size >= MAX
        }
        int getSize()
        {
            return size
        }

        // Function to display all elements in the stack
        void display()
        {
            if (isEmpty())
            {
                print "Stack is Empty"
                return
            }

            print "Stack elements: "
            Node* current = top
            while (current != nullptr)
            {
                print current->data
                current = current->next

                if (current != nullptr)
                {
                    print " -> "
                }
            }
            print newline
        }
};
```

Recursion

Recursion is the technique of making a **function call itself**. This technique provides a way to break complicated problems down into simple problems which are easier to solve. Any method that implements Recursion has two basic parts:

I. Method call which can call itself i.e. **recursive call**
II. A precondition that will stop the recursion which is called **base condition**.

Note that a precondition is necessary for any recursive method as, if we do not break the recursion then it will keep on running **infinitely** and result in a stack overflow. The general syntax of recursion is as follows:

Sample Code:

```
methodName (T parameters…)
{
 if (precondition == true)  //precondition or base condition


{
return result;
}
return methodName (T parameters…);  //recursive call
}
```

Base Condition in Recursion

Sample Code:

```
int Funct(int n)
{    if (n < = 1) // base case       return 1;
    else
        return Funct (n-1);
}
```

Key Points: In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

```
void printAge(int n) {
    if (n < 18) {
        cout << "Age under 18: " << n << endl;
        n += 5; // Increment the age for the next iteration
        printAge(n); // Recursive call
    }
```

Direct and Indirect Recursion

Direct recursion occurs when a function calls itself directly. This is the most straightforward form of recursion, where a function solves a smaller instance of the same problem by repeatedly invoking itself until a base condition is met.

Sample Code (Direct Recursion)

```
void X()
{    // Some code....
    X();
    // Some code...
}
```

Indirect recursion happens when a function calls another function, which in turn calls the original function. This creates a recursive cycle between multiple functions.

Sample Code (In-Direct Recursion):

```
void printAge(int n); // Forward declaration
void incrementAndPrintAge(int n) {
    if (n < 18) {
        cout << "Age under 18: " << n << endl;
        n += 5; // Increment the age for the next iteration
        printAge(n); // Indirect recursive call
    }
}
void printAge(int n) {
    if (n < 18) {
        cout << "Age under 18: " << n << endl;
        n += 5; // Increment the age for the next iteration
        incrementAndPrintAge(n); // Indirect recursive call
   }
}
```

Tailed and Non-Tailed Recursion

We refer to a recursive function as **tail-recursion** when the recursive call is the **last thing** that function executes, so there is nothing left to execute within the current function. The function returns the result of the recursive call directly without any further computation.

Sample Code (Tailed Recursion):

```
void tailRecursion(int n) {
    if (n == 0) {
        return;
    }
    cout << n << " ";
    tailRecursion(n - 1); // Recursive call is the last action (tail call)
}
void Funct (int a)
{
    if (a < 1)  return;// base case
    cout<<a;
    funct (a/2);     // recursive call


}
```

In **non-tail recursion**, the recursive call is not the last operation in the function. There are additional operations performed after the recursive call returns. This type of recursion requires the function to keep track of previous states, leading to more stack space usage.

Sample Code (Non tailed Recursion):

```
void nonTailRecursion(int n) {
    if (n == 0) {
        return; // Base case
    }
    nonTailRecursion(n - 1);
    cout << n << " "; // Additional operation after recursive call
}
void Funct (int a)
{
    if (a < 1)  return; // base case
    return funct (a/2); // recursive call
    cout<<a;
}
```

Dry Run the Following Sample Code:

```
int Funct(int n){
      if(n==0){//base case
            return 1;
      }
      else{
            return n*Funct(n-1);
            cout<<"n: "<<n<<endl;
      }
}
```

Nested Recursion

Nested recursion occurs when a recursive function calls itself with a recursive call as one of its arguments. In other words, the **input parameter** of the recursive call is the **result** of another recursive call.

Sample Code:

```
#include <iostream>
using namespace std;
int fun(int n)
{
    if (n > 100)
        return n - 10;
// A recursive function passing parameter as a recursive call or recursion
inside the recursion
    return fun(fun(n + 11));
}
int main()
{
    int r;
    r = fun(95);
    cout << " " << r;
    return 0;
}
```

Issues in Recursion

1. **Complexity of Recursive Logic:** Designing and understanding recursive solutions can be complex, especially for **problems with multiple recursive calls** or complex base cases.

2. **Stack Overflow:** Recursion heavily relies on the call stack. If the recursion goes too deep, it can cause a stack overflow, leading to a program crash.
   Example:
```
void infiniteRecursion() {
            infiniteRecursion(); // Calls itself indefinitely
}
```

3. **Infinite Recursion:** If the base condition is not defined or incorrect, the recursion may continue indefinitely, causing infinite recursion.
   Example:
```
void badRecursion(int n) {
    if (n > 0) {
        badRecursion(n); // Missing decrement of n
    }
}
```

4. **Performance:** Recursion can be inefficient for large inputs due to repeated calculations. Memorization or iteration may be better for performance in such cases.

   Example: Calculating Fibonacci numbers using simple recursion leads to exponential time complexity.
```
int fibonacci(int n) {
    if (n <= 1) return n; // Base condition
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive calls
}
```

Lab Tasks

Q1: Implement Stack with Linked List:

1. Insert 10 Integers values in the stack
2. Write a utility function to display all the inserted integer values in the linked list in forward and reverse direction both
3. Write utility function to pop top element from the stack

Q2: Implement and insert the values "BORROWROB" in the stack and identify if it's a palindrome or not. Use the push and pop functions to accomplish this (Note: Use Arrays to accomplish this)

Q3: Write a program to reverse a given string using a stack. The program should push each character of the string onto the stack and then pop them off to print the reversed string.

Q4: Implement a program to reverse an array of integers using a stack. The program should push each element of the array into the stack and then pop them off to display the array in reverse order.

Q5:

a. Generate the following sequence with recursive approach

   1 , 3 , 6 , 10 , 15 , 21 , 28 . . . .

b. Generate the following sequence with recursive approach

   0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , 89 , 144 . . .

c. Write an indirect recursive code for the above part (b) with same approach as defined in the In-Direct Recursion.

Q6: Sort The Unsorted Numbers with both tail recursive and Normal recursive approach
Sample Input and Output

Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13

Q9: Write a recursive function to calculate the factorial of a number. The function should stop the recursion when it reaches the base condition, which is when the number becomes 0 or 1.

Q10: Implement a recursive function to compute the nth Fibonacci number using direct recursion.

Q11: Design two functions, A and B, where A calls B and B calls A indirectly, to simulate the behavior of indirect recursion. For instance, function A can print even numbers, and function B can print odd numbers up to a given limit.

Q12: Create a tail-recursive function to compute the greatest common divisor (GCD) of two numbers.

Q13: Implement a recursive function that calculates the result of f(n) = f(f(n - 1)) until n <= 1, where f(0) = 1 using nested recursion.

Q14: Assume you are developing a program to simulate a game where players take turns to guess a number between 1 and 100. The program generates a random number between 1 and 100 at the start of the game, and the player who guesses the correct number wins the game. If a player guesses incorrectly, the program tells them whether their guess was too high or too low, and the turn passes to the next player. Solve it using Direct Recursion?

Q15: Write a C++ function to find the length of a singly linked list using tail recursion?

Q16: Create a C++ program to search for a value in a singly linked list, using non-tail recursion?