# CSA0389 - DATA STRUCTURE FOR STACK IMPLEMENTATION

# ASSIGNMENT - 2

DATE - 31/07/24

BHUVANESHWARI. S
192311375
BE - CSE

20, 10, 40, 50, 90, 30 . delete '20'.

40

Describe the Abstract data type (ADT) and how they differ from concrete data structures. Design an ADT for a stack and implement it using arrays and linked list in c. Include operations like Push, Pop, Peek, Is empty, Is full and Peek.

## Abstract Data Type (ADT):

An Abstract Data Type (ADT) is a theoretical model that defines a set of operation and the behavior of those operations on a data structure, without specifying how the data structure should be implemented. It provides a high level description of what operations can be performed on the data and what constraints apply to those operations.

## Characteristics of ADT's:

* operations: Defines a set of operations that can be performed on the data structure.
* semantics: specifies the behaviour of each operation.
* Encapsulation: Hides the implementation details, focusing on the interface provided to the user.

## ADT for stack:

A stack is a funclamental data structure that follows the Last In, First out (LIFO) principle. It supports the following operations.

Push: Adds an element to the top of the stack.

Pop: Removes and returns the element from the top of the stack.

Peek: Return the element from the top of the stack without removing it.

isEmpty: checks if the stack is empty

isFull: checks if the stack is full.

Concrete data structure:

The implementation using arrays and Linked lists are specific ways of implementing the stack ADT in C.

How ADT differ from concrete data structures:

ADT focuses on the operations and their behavior, while concrete data structures focus on how these operations are realized using specific programming constructs (arrays are linked lists).

Advantages of ADT:

By separating the ADT from its implementation, you achieve modularity, encapsulation and flexibility in designing and using data structures in programs. This separation allows for easier maintenance, code reuse and abstraction of the complex operations.

Implementation in C using Array:

```
#include <stdio.h>
int main() {
    stackArray stack;
    stack.top = -1;
    stack.items[++stack.top] = 10;
    stack.items[++stack.top] = 20;
    stack.items[++stack.top] = 30;
```

```
if (stack.top != -1) {
    printf("Top element: %d\n", stack.items[stack.top]);}
else {
    printf("stack is empty:\n");}
if (stack.top != -1) {
    printf("popped element: %d\n", stack.items[stack.top--]);
  }
else {
    printf("stack underflow:\n");
    }
if (stack.top != -1){
    printf("popped element: %d\n", stack.items[stack.top--]);
} else {
    print("stack underflow:\n");
  }
if (stack.top != -1) {
    printf("Top element after pops: %d\n", stack.items[stack=
                                                    top]);}
else {
    printf("stack is empty:\n");
}
return 0;
}.
```

## Implementation in c using Linked list:

```
# include < studio.h>
int main(){
    Node *top = NULL;
    Node* newnode = (Node*) malloc (sizeof (Node));
if (newNode == NULL){
    printf ("Memory allocation failed: \n");
    return 1;
  }
newNode → data = 10;
new Node → next = top;
top = newNode;
 nownode = (Node*) malloc (size of (Node));
if (nownode == NULL){
   printf ("Memory allocation failed: \n");
   return 1;
}
top = top → next;
Free (temp);
} else{
   printf ("stack underflow: \n");
if (top != NULL){
   printf[" Top element after pops: %d \n", top → data); }
else{
   printf[" stack is empty: \n");}
while (top != NULL){
    Node * temp = top;
    top = top → next;
    free (temp);
}
return 0;
}
```

The university announced the selected candidates register numbers for placement training. The student xxx, reg.no. 20142010 wishes to check whether his name is listed or not. The list is not sorted in any order.

## Linear Search:

Linear search works by checking each element in the list one by one until the desired element is found or the end of the list is reached. It's a simple searching technique that doesn't require any prior sorting to the data.

## Steps for Linear search:

1) start the first element
2) check if the current element is equal to the target element.
3) If the current element is not the target, move to the next element in the list.
4) Continue this process until either the target element is found or you reach the end of the list.
5) If the target is found, return its position. If the end of the list is reached and the element has not been found, indicate that element is not present.

## Procedure:

Given the list:

1) start at the first element of the list.
2) compare '20142010' with '20142015' (first element), 20142033 (second element) these are not equal.

3) compare '20142010' with '20142010' (fifth element).
They are equal.
4) The element '20142010' is found at the fifth position (index 4) in the list.

C code for Linear search:

```
#include <stdio.h>
int main(){
    int regno[] = { };
    int target = 20142010;
    int n = sizeof(regno)/sizeof(regno[0]);
    int found = 0;
    int i;
for (i=0; i<n; i++){
    if (regno[i]==target){
        printf("Registration number %d found at index %d.\n",
                                        target, i);
    found = 1;
    break;
    }
}
if (!found){
    printf("Registration number %d not found in list.\n",
                                        target);
}
return 0;
}
```

Write Pseudocode for stack operations:

1) Push (elements):

```
If stack is full:
    print "stack overflow"
else:
    add element to the top of the stack
    increment top pointer.
```

2) Pop ():

```
If stack is empty:
    print ("stack underflow")
    return null (or appropriate error value)
else:
    remove and return element from the top of stack
    decrement and pointer.
```

3) Peek ():

```
If stack is empty:
    print "stack is empty"
    return null (or appropriate error value)
else:
    return element at the top of the stack (without
                removing it)
```

4) Is Empty ():

```
return true if top is -1 (stack is empty)
otherwise, return false.
```

5) Is Full:

```
return true, if top is equal to maxsize -1 (stack is full)
otherwise, return false.
```

Explanation of Pseudocode:

* Adds an element to the top of the stack. checks if the stack is full before pushing.
* Removes and returning the element from the top of the stack checks if the stack is empty before popping.
* Returns the element at the top of the stack without removing it checks if the stack is empty before peeking.
* checks if the stack is empty by inspecting the top pointer or equivalent variable
* checks if the stack is full by comparing the top pointer or equivalent variable to the max size of stack