NAME : A.Sanjay

REG NO : 192372110

COURSE CODE : CSA0389

COURSE NAME : DATA STRUCTURES

ASSIGNMENT : 01

DATE OF SUBMISSION : 29/4/2024.

Describe the Concept of 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 operations and the Semantics 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 these operations

Characteristics of ADTs :

Operations : Define 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 user.

ADT for Stack :

A stack is a fundamental data structure that follows the Last In, First Out (LIFO) principal. It supports the following operations :

Push : Add element to the top of stack

Pop : Removes and returns the element from top of stack

Peek : Returns the element from top of stack without removing it

IsEmpty : Checks if the stack is empty

IsFull : Checks if the stack is full

Concrete Data Structures :

The implementations using arrays and linked lists are specific ways of implementing stack ADT in C

How ADT differ from concrete data structure.

ADT focuses on the operations and their behaviour while concrete data structures focus on how those operations are realized using specific programming constructs (arrays are linked list)

Advantage of ADT :

By seperating the ADT from its implementation you achieve modularity, encapsulation and flexibility in designing and using data structures in programs.

implementation in c using arrays :-

```
#include <stdio.h>
#define MAX_SIZE 100

typedef struct {
    int items[MAX_SIZE];
    int top;
} StackArray;

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 {
```

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

Implementation in C using Linked List

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int data;
    struct Node *Next;
} Node;
int main() {
    Node *top = NULL;
    Node *newnode = (Node*) malloc(sizeof(Node));
    if(newnode == NULL) {
        printf("memory allocation failed\n");
        return 1;
    }
    newNode -> data = 10;
    newNode -> next = top;
```

```
top=newNode;
newNode = (Node*) malloc(sizeof(Node));
if(newNode==NULL){
        printf("memory allocation failed\n");
        return1;
}
newNode → data=20;
newNode → next=top;
top=newnode;
new node = (Node*)malloc(sizeof Node));
if(newNode==NULL){
        printf("memory allocation failed\n").
        return1;
}
newNode →data=30;
newNode → next=top;
top=newNode;
if(top!=NULL){
        printf("top element:%d\n", top→data);
}else{
        printf("stack is empty!\n");
}
if(top!=NULL){
        Node*temp=top;
        printf("popped element:%d\n", temp→data);
```

```
    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 number 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 order. Identify the searching technique that can be applied and explain the searching steps with the suitable procedure. List includes 20142015, 20142031, 2014 2010, 20142056, 20142003

Linear Search

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

Steps for Linear Search:

→ Start from 1st element

→ Check if the current element is equal to the target element

→ If the current element is not the target move to the next element in the list.

→ Continue this process until either the target element is found or you reach the end of the list.

Procedure:

Given the list:

20142011, 20142033, 20142011, 20142010, 20142056, 20112005

→ Start at the first element of the list

→ Compare '20142010' with '20142011' (first element) '20142033' (second element) '20142011' (third element) these are not equal

→ Compare '20142010' with '20142010' (fifth element) they are equal.

→ The element '2014210' is found at the fifth position (index 4) in the list.

C code for linear search:

```
#include <stdio.h>
int main() {
    int regNumber[] = {20142033, 20142011, 20142010,
                       20142056, 20142003};

    int target = 20142010;
    int n = sizeof(regNumber) / sizeof(regNumber[0]);
    int found = 0;
    int i;
for(i=0; i<n; i++) {
        if(regNumber[i] == target) {
```

```
printf("Registration number: %d found at index %d\n",
                                        target, i);
    found = 1;
    break;
  }
}
if(!found) {
    printf("Reg no: %d not found in list\n", target);
}
return 0;
}
```

Explanation of Code:

→ Reg no array contains the list of registration numbers.

→ target is the registration number we are searching for.

→ 'n' is the total number of elements in array.

→ Iterate through each element of the array.

→ If the current element matches the 'target', print it index and set the 'found' flag to '1'.

→ If the loop completes without finding the target, print that the registration number is not found.

Write pseudocode for stack operations.

Initialize stack():

Initialize necessary variable or structure to represent the stack.

Push:

If stack is full:

print("stack overflow")

else:

add element to the top of the stack

increment top pointer.

Pop():

If stack is empty:

print("Stack underflow")

return null

else:

remove and return element from the top of the stack.

decrement end pointer.

Peek():

If stack is empty:

print("stack is empty")

return null (or appropriate error value)

else:

return element at the top of the stack

otherwise return False

3) full":

return true, if top is equal to max size - 1 (stack is full)

otherwise, return false.

Explanation of Pseudo Code:

* Initialises the necessary variable of data structures to represent a stack

* Adds an element to the top of stack. Checks if the stack is full before pushing.

* Removes and returns the element from the top of the stack.

* Returns the element at the top of the stack without removing it. checks if the stack is empty before peeking