ASSIGNMENT

NAME : B.Lakesh

REG.NO : 192365037

BRANCH : CSE (CS)

SUB : Data Structures for Stack Overflow

SUB CODE : CSA0389

Describe the concept of Abstract data type (ADT) & how they differ from concrete data structures. Design an ADT for a stack & implement it using & linked list in C. Include operations like Push, Pop, Peek, is full, & is Empty.

Sol. **Abstract Data Type:**

An Abstract Data Type is a theoretical model that defines a set of operations & 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 those operations.

Characteristics of ADTs:

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 fundamental data structure that follows the last In, First Out (LIFO) principle. It supports the following operations.

Push: Adds on element to the top of the Stack.

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

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

Is Empty: Checks if the stack is empty.

Is Full: Checks if the stack is full.

Concrete Data Structures:

The implementations using arrays & linked lists are specific ways of implementing the stack ADT in C.

How ADT differ from concrete Data Structures:

ADT focuses on the operations & their behaviour, while concrete data structure focus on how those operations are realized using sepecific programming constructs (arrays are linked lists).

Advantages of ADT:

```
printf ("Stack under flow:\n");
```

By seperating the ADT from its implementation, you archieve modularity, encapsulation, & flexibility in designing & using data structures in programs. This seperation allows for easier maintaince, code reuse & obstraction of the complex operation.

Implementation in C using Arrays:

```
#include <stdio.h>
#define Max_size 100
type def struct {
    int items [MAX_size];
    int top;
} Stack Array;
int main() {
    Stack Array stack;
    stack.top = -1;
    stack.items [++stack.top] = 10;
    stack.items [++stack.top] = 20;
    stack.items [++stack.top] = 30;
if (stack.top != -1) {
  printf ("Topelement: %d \n", stack.items [stack.top]);
  } else {
      print ("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 pops: %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
newNode = (Node *) malloc (sizeof(Node));
if (newNode == NULL) {
    printf("Memory allocation failed: \n");
return 1;
}
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 candiates 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 any order. Identify the searching steps with the suitable procedure. List includes 20142033, 20142015, 20142011, 20142017, 20142010, 20142056, 20142023.

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 pair sorting of the data.

Steps for Linear Search:

1) Start from 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.

Procedure:

Given the list:

'20142033, 20142015, 20142011, 2014, 2017, 20142010, 20142056, 20142023'.

1) Start at the first element of the list.
2) Compare '20142010' with '20142015' (first element), '20142033' (second element), '20142011' (third element), '20142017' (fourth element) these are not equal.
3) Compare '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 regnumbers[7]={20142015, 20142033, 20142011, 20142017, 20142010,
                   20142056, 20142003};
int target = 20142010;
int n = sizeof (regNumbers/sizeof(regNumbers[0]);
int found=0;
int i;
```

```
for (i=0; i<n; i++){
    if (regNumbers[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;
}
```

Explanation of the Code:

1) The 'regNumbers' array contains the list of registration numbers.
2) 'target' is the registration number we are searching for.
3) 'n' is the total number of elements in array.
4) Iterate through each element of the array.
5) If the current element matches the 'target', print its index & set the 'found' flag to '1'.
6) If the loop completes without finding the target, print that the registration number is not found.
7) The program will print the index of the found registration number or indicate that the registration is not present.

Output: Registration number 20142010 found at index 4.

Write Pseudo code for Stack Operations.

1) Intialize stack ():

Initialize necessary variable or structures to represent the stack.

2) Push elements:

```
if stack is full:
    print "Stack Overflow"
else:
    add elements to the top of the stack
    increment top pointer
```

3) Pop():

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

4) 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)
```

5) isEmpty():

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

6) isfull():

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

Explanation of the Pseudocode:

1) Initialize the necessary variables or data structures to represent a stack.
2) Adds an element to the top of the stack. Checks if the stack is full before pusing.
3) Removes & returns the element from the top of the stack. Checks if the stack is empty before popping.
4) Returns the element at the top of the stack without removing it. Checks if the stack is empty before peeking.
5) Checks if the stack is empty by inspecting the top pointer or equivalent variable.
6) Checks if the stack is full by comparing the top pointer or equivalent variable to the maximum size of the stack.