Assignment - 11

Name :- M. Hiteshwar Reddy.

Reg No :- 19232507H

Course Code :- CSA0389

Course Name :- Data structure for stack overflow.

① Describe the concept of Abstract data type (ADT) & 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.

Sol: Abstract Data Type (ADT)

An Abstract Data Type (ADT) is a theoretical model that defines a set of operations and semantics (behaviour) 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 operations that can be performed on 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 fundamental data structure that follows the Last In, First Out (LIFO) Principle. It Supports the following operations:

- Push: Adds an element to top of the stack.
- pop: Removes and returns the element from top of the stack.
- Peek: Returns the element from the top of stack without removing it.
- is Empty: checks if the stack is empty.
- is Full: checks if the stack is full.

Concrete Data structure:-

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

How ADT differs from concrete Data structures.

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

Advantages of ADT:-

By separating the ADT from its implementation, you acheieve modularity, encapsulation and flexi-bility 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 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 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 <stdio.h>
typedef struct Node{
   int data;
   struct Node* next;
 }Node;
int main() {
    Node* top= NULL;
   Node * newNode = (Node *) malloc(size of(Node));
 if (newNode == NULL) {
 Printf("Memory allocation failed!\n");
   return 1;
 }
   newNode -> data = 10;
   new Node -> next = top;
 top = new Node;
new Node = (Node*)malloc(size of(Node));
if (new Node = NULL) {
      Printf("Memory allocation failed\n");
```

```
    return 1;
}
new Node -> data = 20;
new Node -> next = top;
top = new Node;
new Node = (Node*) malloc (sizeof (Node));
if (new Node == NULL) {
    Printf ("Memory allocation failed \n");
return 1;
}
new Node -> data = 30;
new Node -> next = top;
top = new Node;
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) {
```

```
    return 1;
}
new Node -> data = 20;
new Node -> next = top;
top = new Node;
new Node = (Node*) malloc (sizeof (Node));
if (new Node == NULL) {
    Printf ("Memory allocation failed \n");
return 1;
}
new Node -> data = 30;
new Node -> next = top;
top = new Node;
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;
}.
```

(2) The university announced the selected candidates register no. 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 technique that can be applied & explain the searching steps with suitable procedure. List includes 20142015, 20142033, 20142011, 20142017, 20142010, 20142056, 20142003.

(A). Linear Search : Searching technique

Linear search works by checking each element in list one by one until desired element is found. It's simple searching technique that doesn't require any prior sorting of Data.

Steps for linear Search:

1) start from the first element.
2) check if the current element is equal to target element.
3) If current element is not target, move next element in list
4) continue this process until either target element is found.
5) If target is found, return the pos^n. If end of the list is reached and the element has not been found, indicate that element is not present.

Procedure:-

Given the list:

('20142015', 20142033, 20142011, 20142017, 20142010, 20142056, 20142057

1) start at first element of list
2) compare '20142010' with '20142015' (first element), '20142033 (second element) '20142011 (3rd element) '20142017' (fourth element) these are not equal.
3) compare '20142010' with '20142010' (5th element). They are equal.
4) The element '20142010' is found at 5th pos^n (Index 4) in list.

C code for linear search :-

```
#include <stdio.h>
int main() {
    int regNumbers[] = {20142015, 20142033, 20142011, 20142017,
                        20142010, 20142056, 20142033};

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

Explation of code :-

1) The 'reg No' array contains the list of registration numbers.

2) 'target' is the registration No we are searching for

4) Iterate through each element of array.

5) If loop completes without finding the target, Print that registration no. is not found.

6) Program will Print index of found registration

Output:- Registration number 20142010 found at index 4.

3) Write Pseudo code for stack operations.

① Intialize stack ():

Intialize necessary variable or structures to represent the stack.

② staPush (element):

if stack is full:
    Print "stack overflow".
else:
    add element to top of stack
    increment top pointer

③ POP ():

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

④ Peek ():

if stack is empty:
    Print "stack is Empty"
    return null (or appropiate error value)
else:
    return element top of stack (without removing)

(5) is empty():
return true: if top is -1 (stack is empty)
Otherwise, return false

(6) is full:
return true, if loop is equal to max size -1 (stack full)
return false,

Explanation of the Pseudocode:-

- Initialize the necessary variables or data structures to represent a stack.
- Adds an element to top of stack. checks if the stack is full before Pushing.
- Returns the element at top of stack without removing it checks if stack is empty before peeking.
- checks if stack is full by comparing the top pointer or equivalent variable to maximum size of stack.