NAME : J. RAMESH

REG NO : 192311230

COURSE CODE : CSA-0389

COURSE NAME : Data structure for stack overflow

ASSIGNMENT : 02

Date of submission : 29-07-2024

① Describe the concept of 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.

Sol-

ABSTRACT Data type (ADT) :-

An Abstract Data type (ADT) is a theoretical model that defines a set of operations and the semantics (behaviour) of those operations on a data structure, without specifying how to 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.

Characters 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 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 an element to the top of the stack.
* pop :- Removes and 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 implementing using arrays and linked lists are specific ways for implementing the stack ADT in C.

How ADT differs from concrete data structures

ADT focuses on the operations and their behaviour, while concrete data structures focus on how those operations are related using specific programming constructs.

Advantages of ADT :-

By separating the ADT from its implementations you achieve modularity encapsulation and flexibility in designing and using data structure 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("Poped element : %d\n", stack.item[stack.top--]);
} else {
```

```
    printf("stack underflow:\n");
}
 if(stack.top != -1) {
    printf("top element after pops: %d\n", stack.item);
} 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(sizeof(node));
if (newnode == NULL) {
    printf("memory allocation failed!\n");
    return 1;
}
newnode -> data = 10;
newnode -> next = top;
```

```
top = new node;
newnode = (Node*) malloc (size of (Node));
if (newnode == NULL) {
    printf ("memory allocation failed \n");
    return 1;
}
newNode -> data = 20;
newNode -> Next = top;
top = newNode;
newNode = (Node*) malloc (size of (Node));
if (newNode == NULL) {
    printf ("memory allocation failed :(\n");
return 1;
}
new node -> 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 pop: %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 any order identify the searching technique that can be applied and explain the searching steps with the suitable procedure. list include 20142015, 20142033, 20142011, 20142017, 20142010, 20142056, 20142003.

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 is a simple searching technique that doesn't require any prior sorting of the data.

steps for linear search :-

* start from the first element
* check if the current element is equal to the target element.
* if the current element is not the target element is found or you reach.
* continue this process until either the target elements is found or you reach the end of the list.
* if the target is found, return its position. if the end of the list is reached and the element has not been found indicate the element is not present.

- Procedure :-

Given the list :

20142015, 20142033, 20142011, 20142017, 20142010, 20142056

20142002.

* start the first element of the list
* compare '20142010' with 20142015 (first element) 20142033 (second element). '20142011 (third element) 20142017 (fourth element)

these are not equal.

* compare '20142010' with '20142010' (fifth element) they are equal.
* the element '2014210' is found at the fifth position index in the list

C code for linear search :-

```
# include <stdio.h>
int main () {
    int regNumbers[] = { 20142015, 20142033, 20142011, 20142017
                         20142010, 20142056, 20142003 };

    int target = 20142010;
    int n = size of (regNumbers) / size of (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 :-

* the reg numbers array contains the list of registration numbers
* target is the registration number we are searching
* 'n' is the total number of elements in array.
* iterate through each element of the array.
* if the current element matches the target, print its index and set the found flag to '1'.
* if the loop completes without finding the target print that the registration number is not found
* 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

(3) Write pseudocode for stack operations

Ans

Initialize stack():

Initialize necessary variable or structures to represent the stack.

(2) push(element):

if stack is full:

print "stack overflow"

else:

add element 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 and 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.

(5) is empty():

return true if top is -1 (stack is empty)

otherwise, return false

⑥ is full

return true, if top is equal to max size - 1
otherwise, return false.

Explanation of the pseudocode :-

* initializes the necessary variables or data structures to represent a stack.
* adds an element to the top of the stack, checks if the stack is full before pushing.
* removes and returns 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 maximum size of the stack.