ASSIGNMENT-2

NAME : A. Nagaveni

REG.NO : 192311179

SUB.CODE : CSA0389

SUBJECT : DATA STRUCTURE

DATE OF SUBMISSION : 31-07-2024.

① Describe the concept of Abstract data type (ADT) & how they differ from concrete data structures Desig-n on ADT for a stack and implement it using arrays and linked list inc. 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 (behaviour) of those operations on a data structure should without specifying how the data structure should be implemented. It provides a high level description of what operations can be performed on the data & what constrains apply 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 toup of the stack.
- Pop : Removes and returns that Element. from the top of the stack.

- peek : Returns the Element from the top of the st[illegible] without removing it.

- isempty : checks if the stack is Empty.
- is Full : checks if the stack is full.

Concrete Data structures :-

The implementations using arrays & lin[illegible] 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 structures focus on how those operations are realised using specific programm[illegible] constructs (arrays are linked lists).

Advantages of ADT :-

By seperating the ADT from its imple[illegible]-ntation, you achieve modularity, Encapsulation and flexibility in designing & using data structures in programs, this separation allows for Easier maintenan[illegible] code reuse and abstraction of the complex operatio[illegible]

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 ("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-1]);
} else {
    printf ("stack underflow!\n");
}
if (stack.top!=-1){
    printf ("popped element : %d \n", stack.items
}else {                                  [stack.top--]);

    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 (new Node == NULL) {
    printf ("Memory allocation failed ! \n");
    return 1:
}
New node -> data = 10;
new node -> next = top;
top = newnode;
newnode = (Node *) malloc (size of (Node));
if (new Node == NULL){
    printf (" Memory allocation failed \n");
    return1;
}
```

```
newNode -> data = 20;
new Node -> next = top;
  top = new Node;
new Node = (Node*) malloc (size of (Node));
if (new Node == NULL){
   printf ("Memory allocation failed ! \n");
  return;
}
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 is Empty! \n");
}
 while (top != NULL) {
          Node* temp = top;
          top = top -> Next;
          free (temp);
}
return 0;
}
```

2) The university announced the selected candiates register number of placement training. The student xxx, reg no. 2014 2010 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 includes 2014 2015, 20142033, 20142011, 20142017, 20142010, 20142036, 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's a simple searching technique that doesn't require any prior sorting of the data.

Steps for linear search:-

1) start from the 1st element.
2) check if the Element is = the target ele.
3) If cur-Ele isn't the target, move to the next Element
4) continue the process until either the targ-Ele is found.
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 :-

20142015, 20142033, 20142011, 20142017, 20142010, 20142036.

1) start at the 1st element of the list
2) compare '20142010' with '20142025' (1st element), 20142033 (2nd ele), '20142011' (third element), '20142017' (fourth ele) these are not equal.
3) compare '20142010' with '20142016' (5th ele) they are equal.
4) The element '20142010' is found at the 5th position index.
5) 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=sizeof(regnum]/sizeof(regnum[0]);
int found =0;
int i;
for (i=0; i<n; i++){
  if (regnum [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 \n",
```

```
        target);
}
return 0;
}.
```

Explanation of the code :-

① The 'reg numbers' array contains the list of registra-
-tion 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 and set the 'found' flag to '1'

⑥ If the loops completes without finding the target, print that the registration number is not found.

⑦ The program will print the index of the found registration, now indicate that the registration is not found.

Output :- Registration number 2014206 found at index 4.

Write Pseudocode for stack operations?

1) Intialize stack()

Intialize 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 & return element from the top of the stack.

decrement and 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) 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 (stack is full)

otherwise, return false.

Explanation of the pseudocode:-

- Intialize 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 & 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 stacks 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.