Assignment - 2

CSA0389 - Data Structure for stack implementation.

Name : Priyadharshini. B

Reg NO : 192311337

Date : 30/07/24

Topic : pseudocode for stack operation.

Describe the concept of Abstract data type (ADT) and how they differ from conmete data structures. Design on ADT for a stack and Imple ment it using arrays and linked list in c Include operations like Push. Pop, peek, is empty, is full and peek

ABSTRACT DATA TYPE :

An Abstrate data type is a theoretical model that defines a set of operations and the 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 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.

semanties: 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 stack of top

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

is empty: check if the stack is empty

is full : check if the stack is full .

Concrete data Structures :

The implementations using arrays and linked lists are specific ways of Implementing the 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 these operations are realised using specific programming constructs (arrays are linked list)

ADV OF ADT :

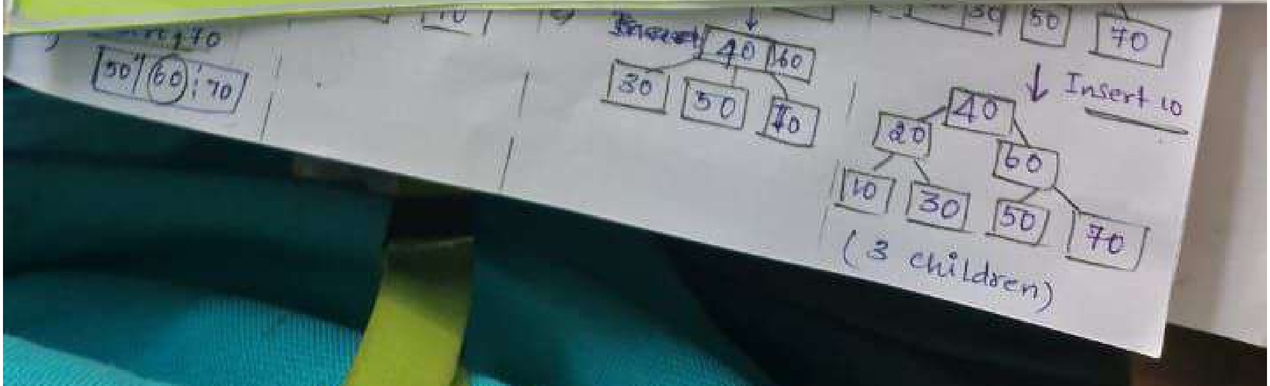
By separating from its implementation system , you achieve modularity , encapsulation , and flexibility in designing and using data structures in programs. This separation allows for easier maintanance , code reuse and abstruction of the complex operations .

Implementation in c using Arrays :

```
#include <stdio.h>
#define MAX SIZE 100
type def struct {
    int items [MAX_SIZE] ;
    int top ;
}stackArray ;
int main () {
    stack Array stack ;
    stack.top = -1 ;
    stack.items [++stack.top]=10 ;
    stack.items [++stack.top] =20 ;
    stack.items [++stack.top] =30 ;
    printf ("memory allocation failed \n").
 return 1 ;
 }
 newNode -> data = 30 ;
 newNode -> next = top ;
  top = newNode ;
 if (top != NULL) {
       printf (" rop element : %d\n", top->data) ;

 }
 else {
      printf ("stack is empty :\n");
}
if (top : = NULL) {
       Node *temp = top ;
       printf ("popped element : %d\n",temp->data ;
       printf ("stack underflow : \n")
  }
 if (stack.top != -1) {
       printf ("rop element after pops = %d\n",stack.items (stack.top) ;
 }
 return 0 ;
}

        # include <stdio.h>
        # include <stdlib.h>
        typedef struct Node {
             int data ;
```



```
  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) {
    print ["popped element : %d\n", stackitems (stack . top = 1));
 }
 else {
     top = newNode ;
    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) {
```

3) 50 60 70

```
        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;
   newnode->next=top.
   top.top->next
    free(temp);
 }
 else{
    printf("stack underflow :\n");
while(top!=null){
   node *temp=top;
   top.top->next ;
   free(temp);
}
return 0;
}
```

tree.

to Reduce the number of disc acces

* Time complexity to insert, delete - O(logn).

* All leaves are at same value.

M-1 key values :

(ii) 2-3 TREE : Insert 50, 60, 70, 40, 30, 20. → (each node has 2 or 3 chidren 4 atmost ... val

Types of ...

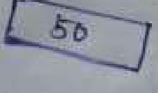
i) 2-3 Tr...

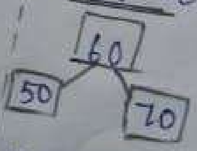
ii) 2-~~3~~-4

iii) 2-3-4-...

iv) 2-3-4-5...

1) Insert 50





5) Insert 30

(6) Insert 20

2) Insert ...

4) Insert 40

1) Intialize stack ( ):

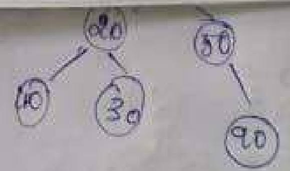
Intialize necessary variable or structure to present the stack.

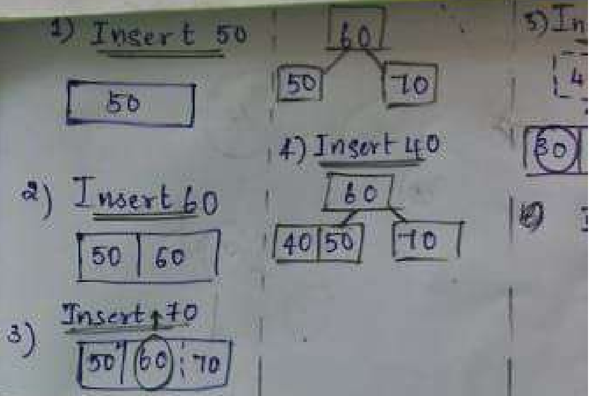
```
2) push (elements):
        if stack is full:
            print "stack overflow"
        else:
            add element to the top of the stack
            increment top pointer

3) pop ():
        if stack is empty:
            print ("stack underflow")
            return null (or appropriate errorvalue)
    else:
        remore and return the element to the top of the stack
        decrement end pointer

4) peek():
    if stack is empty():
        print ("stack is empty")
        return null or appropriate errorvalue

5) isempty():
    return true if Top is -1 (stack is empty)
```





6) Is full :

return True , if top Is equal to maxsize -1 (stack is full) otherwise , return False .

Initialize the necessary varaables or data structure to represent a stack

Adds an element to the top of the stack , check if the stack is full before pushing

Remores and returns the element from the top of the stack . check if the stack is empty before popping .

Returns the element at the top of the stack without removing it . check if the stack is empty without peeking .

checks if the stack is empty by Inspecting the top painted or equivalent variable

checks if the stack if full by comparing the top pointer or equivalent variable to the maximum size of the stack .

linear search works by checking each element in the list one by one until the desired element found in the end of the list that doesn't require any prior sorting of the data .

50 | 60 : 70

10 | 30 | 50 | 70

(3 children)

```
#include <stdio.h>
int main (){
 int regNumbers = {20142010};
 int target =20142010 ;
 int n = sizeof(regnumbers /sizeof (regnumber(0));
 int found =0 ;
 int ;
 for (i=0; i<n ; i++){
   if (regnumbers(i)=target) {
 printf ("Registration number is found at index :%d\n",
                                        target,i) ;
 found =1 ;
   break ;
  }
}
if (!found) {
 printf ("Registration number %d not found in
                  list  :\n",target ) ;
 }
  return0 ;
 }.
```