Name :- CH. Nagi Reddy

Register No :- 192324170

Course Code :- CSA0389

Course Name :- Data Structure

Assignment - 2

Date :- 31/07/2024

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 (behavior) of those operations on a datastructure, without specifying how the data structure should be implemented. It provides 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 behavior of each operation.
* Encapsulation:- Hide 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 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 and linked lists are specific ways of implementing the stack ADT in C.

How ADT differ from Concrete Data structures:

ADT focuses on the operations and 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 achieve modularity, encapsulation, and flexibility in designing and using datastructures in programs. This separation allows for easier maintanance, code reuse, and abstraction 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() {
    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 ("poped element: %d \n", stack.items[stack.top--]);
```

```
}else {
    printf ("Stack underflow! \n");
}
if (stack.top!=-1){
   printf ("poped 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 ~~using~~ 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");
        return 1;
```

```
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;
newNode = (Node*) malloc (sizeof (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;
}
```

The university announced the selected candidates register numbers for placement training. The student 'xxx', reg.no. 2014xxxx 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 technique that can be applied and explain the searching steps with the suitable procedure. List includes 2014xxxx, 2014xxxx, 2014xxxx, 2014xxxx, 2014xxxx, 2014xxxx, 2014xxxx.

Sol:

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 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 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 target element is found or you reached the end of the list.
5. If the target is found, returns 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:

'2014xxxx, 2014xxxx, 2014xxxx, 2014xxxx, 2014xxxx, 2014xxxx, 2014xxxx.'

1. Start at the first element of the list.
2. Compare '2014xxxx' with '2014xxxx' (first element), '2014xxxx' (second element), '2014xxxx' (third element), '2014xxxx' (fourth element) - these are not equal.
3. Compare '2014xxxx' with '2014xxxx' (fifth element). - they are equal.
4. The element '2014xxxx' is found at the position (index 11) in the list.

C Code for Linear Search:

```
#include <stdio.h>
int main() {
    int regnumbers[] = {20142015, 20142033, 20142011, 20142017, 20142010, 20142005,
                        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 and 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 pseudocode for stack operations.

1. Initialize stack():
   Initialize necessary variable or structures to represent 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 error value)
   else:
      remove and return element from 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:

* Initialize 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 poping.
* 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.