Assignment - 1

Name : R. Nikhila

Registernumber : 192372186

Course Code : CSA0389

Course Name : Data structure for stack overflow

Submittion Date : 31-07-2024

No of pages : 12

Faculty Name : Dr. Ashok kumar

Assignment number : 01

Department, year : CSE(AI), 2nd year

1) Describe the concept of abstract data type (ADT) and how they differ from concrete data structure. 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 ADT is a theoritical model that defines a set of operations and the semantics (behavior) of those operation 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 constraint apply to those operations.

Characteristic of ADT:

- Operations: Define a set of operations that can be performed on data structure
- Semantics: specifies the behavior of each operation
- Encapsulation: Hides the implementation details, focusing on the interface provided to 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: Add an element to the top of stack

pop: Removes and returns the element from top of stack

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

is empty: checks if stack is empty

is full: checks if stack is full

Concrete Data structure:

The implementation using array and linked lists are specific ways of implementing the stack ADT in C

How ADT differ from Concrete Data structure:

ADT focuses on operations and their behavior, while concrete data structures focus on how those operations are realised 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 data structures in program. this separation allows for easier maintenance, code reuse, and abstraction of the complex operations.

Implementation in C using array:

```
#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("popped element: %d\n", n);
}
 if (stack.top != -1) {
     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 pop: %d\n", stack.items[stack.
}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 (new node == NULL){
    printf("memory allocation failed!\n");
    return 1;
}
new node -> 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;
}
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 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 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 candidate 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 suitable procedure. list includes 20142015, 20142033, 20142011, 20142017, 20142010, 20142056, 20142003.

A) linear search:

linear search works by checking each element in 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 data

Steps of linear search:

1) start from first number

2) check if the current element is equal to target element

3) If the current element is not target, move to next element in list

4) continue this process until either target element is found or you reach the end

5) If the target is found, return its position. If the end of list is reached and the element has not been found, indicate that element is not present.

Procedure:

1) Start at the first element of list.
2) Compare 20142010 with 20142015. these are not equal
3) Compare 20142010 with 20142010. They are equal
4) The element 20142010 is found at the fifth position (index in the list)

C code for linear search:

```
# include <stdio.h>
int main () {
int rg num [] = {20142015, 20142033, 20112011,
  20142017, 20142010, 20142056, 201420034};
int target = 20142010;
int n = size of (rg num) / size of (rg num (0));
  int found = 0;
    int i ;
    for (i=0; i<n; i++)
    {
    if (rgnum (i) == target)
     {
```

```
printf("Regista number found at index %d\n", target);
    found = 1;
    break;
  }
 }
if (!found) {
printf("registration number %d not found in list);
}
return 0;
}
```

Explanation of the code:

1) the regnum array contains the list of registration num
2) Target is registration number we are searching for
3) n is total number of elements in array
4) iterate through each element of the array
5) If current element matches the target, print it index and set the found flag to 1
6) If the loop completes without finding the target print that number is not found
7) the program will print the index of found num & indicate is not present

Output: Registration number 20142010 found at index 4

3) write pseudocode for stack operations.

1) Initialize stack ():
Initialize necessary variable & structures to represent the stack

2) push 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"
else:
add element to top of stack
increment top pointer
```

4) peek():

```
if stack is empty:
print "stack is empty"
return null
else:
return element at top of stack
```

5) is empty.is:

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:

1) Initialize the necessary variable & data structure to represent a stack.

2) Add an element to top of stack - checks if stack is full before pushing.

3) Removing and returns the element from the top of the stack checks if stack is empty before popping

4) Returns the element at the top of stack without removing it. checks if the stack is empty before peeking

5) checks if stack is empty by inspecting the top pointer & equivalent variable

6) checks if stack is full by comparing the top pointer & equivalent variable to maximum size of stack