Stacks

In this lecture, the following topics are covered:

- ❑ The concept of Stack
- ❑ Stack Representation using Arrays
- ❑ Stack Representation using Linked Lists

1. The Concept of Stack

- Stack is an important data structure which stores its elements in an ordered manner.
- We will explain the concept of stacks using an analogy.
- You must have seen a pile of plates where one plate is placed on top of another as shown in Fig. 7.1.
- Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

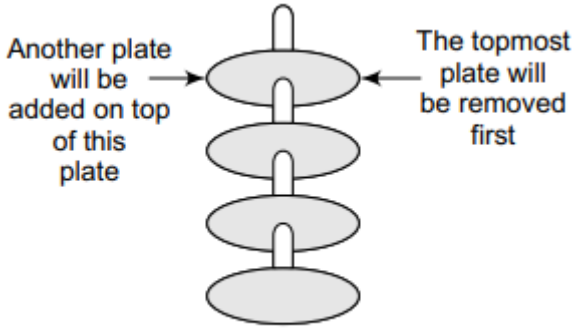

Figure 7.1 Stack of plates

- A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP.
- Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.
- Now the question is where do we need stacks in computer science? The answer is in function calls. Consider an example, where we are executing function A. In the course of its execution, function A calls another function B. Function B in turn calls another function C, which calls function D.

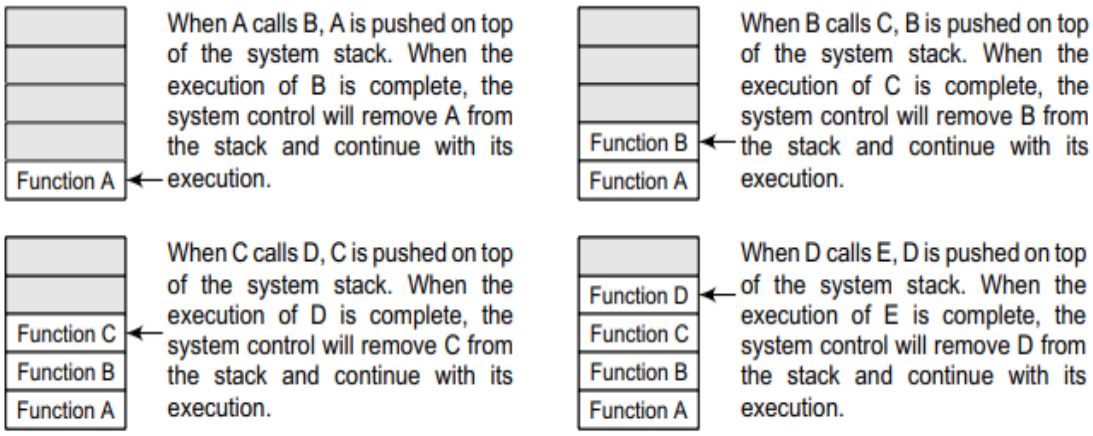

Figure 7.2 System stack in the case of function calls

- In order to keep track of the returning point of each active function, a special stack called system stack or call stack is used.
- Whenever a function calls another function, the calling function is pushed onto the top of the stack. This is because after the called function gets executed, the control is passed back to the calling function.
- Look at Fig. 7.2 which shows this concept.
- Now when function E is executed, function D will be removed from the top of the stack and executed.
- Once function D gets completely executed, function C will be removed from the stack for execution.
- The whole procedure will be repeated until all the functions get executed.
- Let us look at the stack after each function is executed. This is shown in Fig. 7.3.
- The system stack ensures a proper execution order of functions.
- Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled.
- Stacks can be implemented using either arrays or linked lists. In the following sections, we will discuss both array and linked list implementation of stacks

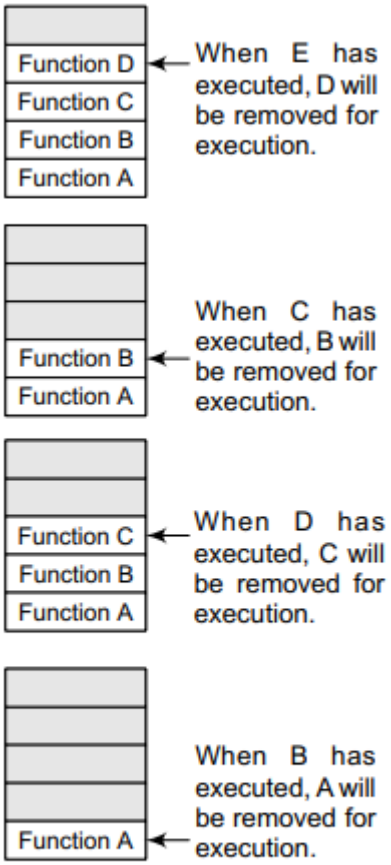

Figure 7.3 System stack when a called function returns to the calling function

2. An Array Representation of Stacks

- In the computer's memory, stacks can be represented as a linear array.
- Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.
- There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.
- If TOP = NULL, then it indicates that the stack is empty and
- if TOP = MAX–1, then the stack is full. (You must be wondering why we have written MAX–1. It is because array indices start from 0.) Look at Fig. 7.4.

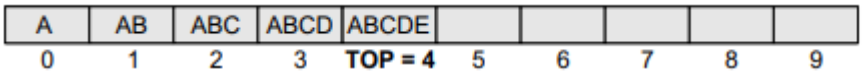

Figure 7.4 Stack

- The stack in Fig. 7.4 shows that TOP = 4, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

2.1 Operations on Stacks

- A stack supports three basic operations: push, pop, and peek.
- The push operation adds an element to the top of the stack and
- the pop operation removes the element from the top of the stack.
- The peek operation returns the value of the topmost element of the stack.

2.1.1 Push Operation

- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if TOP=MAX–1, because if that is the case, then the stack is full and no more insertions can be done.

- If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed. Consider the stack given in Fig. 7.5.

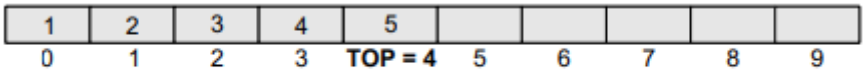

Figure 7.5 Stack

- To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP]. Thus, the updated stack becomes as shown in Fig. 7.6.

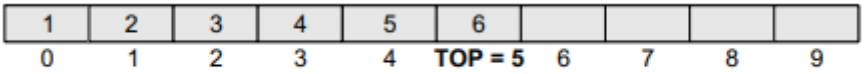

Figure 7.6 Stack after insertion

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Figure 7.7 Algorithm to insert an element in a stack

- Figure 7.7 shows the algorithm to insert an element in a stack. In Step 1, we first check for the OVERFLOW condition. In Step 2, TOP is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed by TOP.

2.1.2 Pop Operation

- The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack given in Fig. 7.8

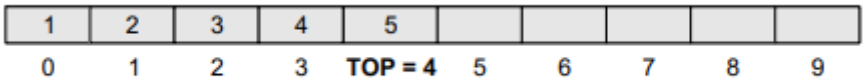

Figure 7.8 Stack

- To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP. Thus, the updated stack becomes as shown in Fig. 7.9.

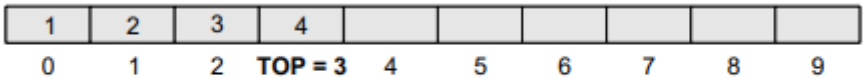

Figure 7.9 Stack after deletion

- Figure 7.10 shows the algorithm to delete an element from a stack. In Step 1, we first check for the UNDERFLOW condition. In Step 2, the value of the location in the stack pointed by TOP is stored in VAL. In Step 3, TOP is decremented.

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Figure 7.10 Algorithm to delete an element from a stack

2.1.3 Peek Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Figure 7.11 Algorithm for Peek operation

-
- The algorithm for Peek operation is given in Fig. 7.11. However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.

- Consider the stack given in Fig. 7.12

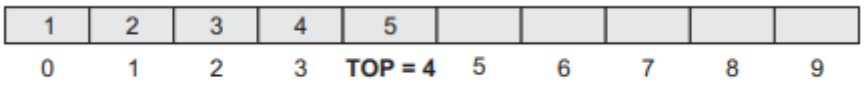

Figure 7.12 Stack

- Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

The following **C** program shows how to implement a stack represented by an array:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

void push (int stack[], int value);
int pop (int stack[]);
int peek (int stack[]);
void display(int stack[]);
bool isEmpty();
bool isFull(int size);
int top = -1;

int main()
{
    int n = 0;
    printf("\n Enter the size of stack: ");
    scanf("%d", &n);
    int stack[n];

    int val, option;
```

```c
do
{
    printf("\n *****MAIN MENU*****");
    printf("\n 1. PUSH");
    printf("\n 2. POP");
    printf("\n 3. PEEK");
    printf("\n 4. DISPLAY");
    printf("\n 5. EXIT");
    printf("\n Enter your option: ");
    scanf("%d", &option);
    switch(option)
    {
    case 1:
        if(!isFull(n)){
            printf("\n Enter the number to be pushed on stack: ");
            scanf("%d", &val);
            push(stack, val);
        }
        else{
            printf("Sorry, stack is full!\n");
        }
        break;


    case 2:
        if (!isEmpty()){
            val = pop(stack);
            printf("\n The value deleted from stack is: %d", val);
        }
        else{
            printf("Sorry, stack is empty!\n");
        }

        break;
    case 3:
        if (!isEmpty()){
            val = peek(stack);
             printf("\n The value stored at top of stack is: %d", val);
            }
           else{
            printf("Sorry, stack is empty!\n");
        }
        break;
    case 4:
        display(stack);
        break;
     }
}while(option != 5);
    return 0;
}
```

```
void push (int stack[], int value){

    stack[++top] = value;
}
int pop (int stack[]){
    return stack[top--];
}
int peek (int stack[]){
    return stack[top];
}
void display(int stack[]){

    while (top != -1){
        printf("%d ", pop(stack));
    }
}
bool isEmpty(){
    return (top == -1);
}
bool isFull(int size){
    return (top == size-1);
}
```

3. A Linked List Representation of Stacks

- We have seen how a stack is created using an array.
- This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size.
- In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.
- But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.
- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node.
- The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.
- The linked representation of a stack is shown in Fig. 7.13.

Figure 7.13 Linked stack

3.1 Operations on A Linked Stack

A linked stack supports all the **three** stack operations, that is, **push**, **pop**, and **peek**.

3.1.1 Push Operation

- The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Fig. 7.14

Figure 7.14 Linked stack

- To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP != NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP. Thus, the updated stack becomes as shown in Fig. 7.15.

Figure 7.15 Linked stack after inserting a new node

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Figure 7.16 Algorithm to insert an element in a linked stack

Figure 7.16 shows the algorithm to push an element into a linked stack. In Step 1, memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. This is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

3.1.2 Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown in Fig. 7.17.

Figure 7.17 Linked stack

In case TOP != NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in Fig. 7.18.

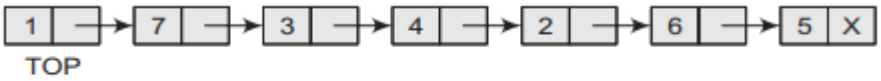

Figure 7.18 Linked stack after deletion of the topmost element

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Figure 7.19 Algorithm to delete an element from a linked stack

Figure 7.19 shows the algorithm to delete an element from a stack. In Step 1, we first check for the UNDERFLOW condition. In Step 2, we use a pointer PTR that points to TOP. In Step 3, TOP is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

The following **C** program shows how to implement a stack represented by a linked list:

PROGRAMMING EXAMPLE

Write a program to implement a linked stack.

```
##include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>
struct stack
{
        int data;
        struct stack *next;
};
struct stack *top = NULL;
struct stack *push(struct stack *, int);
struct stack *display(struct stack *);
struct stack *pop(struct stack *);
int peek(struct stack *);

int main(int argc, char *argv[]) {
        int val, option;
do
{
    printf("\n *****MAIN MENU*****");
    printf("\n 1. PUSH");
    printf("\n 2. POP");
    printf("\n 3. PEEK");
    printf("\n 4. DISPLAY");
    printf("\n 5. EXIT");
    printf("\n Enter your option: ");
    scanf("%d", &option);
    switch(option)
    {
    case 1:
        printf("\n Enter the number to be pushed on stack: ");
        scanf("%d", &val);
        top = push(top, val);
        break;
    case 2:
        top = pop(top);
        break;
    case 3:
        val = peek(top);
        if (val != -1)
        printf("\n The value at the top of stack is: %d", val);
        else
        printf("\n STACK IS EMPTY");
        break;
    case 4:
        top = display(top);
        break;
    }
}while(option != 5);
return 0;
```

```
}
struct stack *push(struct stack *top, int val)
{
        struct stack *ptr;
        ptr = (struct stack*)malloc(sizeof(struct stack));
        ptr -> data = val;
        if(top == NULL)
        {
                ptr -> next = NULL;
                top = ptr;
        }
        else
        {
                ptr -> next = top;
                top = ptr;
        }
        return top;
}
struct stack *display(struct stack *top)
{
        struct stack *ptr;
        ptr = top;
        if(top == NULL)
        printf("\n STACK IS EMPTY");
        else
        {
                while(ptr != NULL)
                {
                        printf("\n %d", ptr -> data);
                        ptr = ptr -> next;
                }
        }
        return top;
}
struct stack *pop(struct stack *top)
{
        struct stack *ptr;
        ptr = top;
        if(top == NULL)
        printf("\n STACK UNDERFLOW");
        else
        {
                top = top -> next;
                printf("\n The value being deleted is: %d", ptr -> data);
                free(ptr);
        }
        return top;
}
int peek(struct stack *top)
{
        if(top==NULL)
        return -1;
        else
        return top ->data;
}
```

4. Applications of Stacks

In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution.

The following are some of the problems where the stack can be applied:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion

However, only reversing a list and recursion are introduced here.

4.1 Reversing a List

A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index. The following program shows how to use the stack data structure to reverse an **n** integer numbers stored in an array. Notice that pushing first these numbers into the stack and then popping them out can accomplish this task.

Write a program to reverse a list of given numbers.

```
#include <stdio.h>
#include <conio.h>
int stk[10];
int top=-1;
int pop();
void push(int);
int main()
{
        int val, n, i,
        arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array : ");
        for(i=0;i<n;i++)
                scanf("%d", &arr[i]);
        for(i=0;i<n;i++)
                push(arr[i]);
        for(i=0;i<n;i++)
        {
                val = pop();
                arr[i] = val;
        }
        printf("\n The reversed array is : ");
        for(i=0;i<n;i++)
                printf("\n %d", arr[i]);
        getche"();
        return 0;
}
void push(int val)
{
        stk[++top] = val;
}
int pop()
{
        return(stk[top--]);
}
```

4.2 Recursion

- Recursion is an implicit application of the stack.
- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.
- Every recursive solution has two major cases. They are:
    - **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
    - **Recursive case**, in which first the problem at hand is divided into simpler sub-parts. Second, the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.
    - Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.
- To understand recursive functions, let us take an example of calculating factorial of a number. To calculate **n!**, we multiply the number with factorial of the number that is 1 less than that number. In other words, n**!** = n × (n–1)**!.**

Let us say we need to find the value of 5!

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$
$$= 120$$

This can be written as

$$5! = 5 \times 4!, \text{ where } 4! = 4 \times 3!$$

Therefore,

$$5! = 5 \times 4 \times 3!$$

Similarly, we can also write,

$$5! = 5 \times 4 \times 3 \times 2!$$

Expanding further

$$5! = 5 \times 4 \times 3 \times 2 \times 1!$$

We know, $1! = 1$

The series of problems and solutions can be given as shown in Fig. 7.27.

PROBLEM	SOLUTION
5!	5 × 4 × 3 × 2 × 1!
= 5 × 4!	= 5 × 4 × 3 × 2 × 1
= 5 × 4 × 3!	= 5 × 4 × 3 × 2
= 5 × 4 × 3 × 2!	= 5 × 4 × 6
= 5 × 4 × 3 × 2 × 1!	= 5 × 24
	= 120

Figure 7.27 Recursive factorial function

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number.

Every recursive function must have a base case and a recursive case. For the **factorial function**:

- **Base case** is when n = 1, because if n = 1, the result will be 1 as 1! = 1.
- **Recursive case** of the factorial function will call itself but with a smaller value of **n**. This case can be given as:

  $$factorial(n) = n \times factorial\ (n-1)$$

  Look at the following program which calculates the factorial of a number recursively

Write a program to calculate the factorial of a given number.

```
#include <stdio.h>
int Fact(int);    // FUNCTION DECLARATION
int main()
{
        int num, val;
        printf("\n Enter the number: ");
        scanf("%d", &num);
        val = Fact(num);
        printf("\n Factorial of %d = %d", num, val);
        return 0;
}
int Fact(int n)
{
        if(n==1)
                return 1;
        else
            return (n * Fact(n-1));
}
```

Output

```
Enter the number : 5
Factorial of 5 = 120
```