LECTURE NOTES

PROGRAMME – BCA

SEMESTER- II

DATA STRUCTURE USING C(BCA 204)

UNIT II

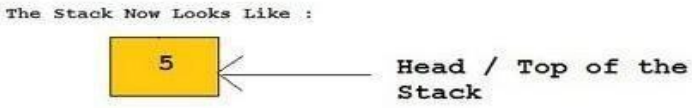
Stack ADT

- Stack is a specialized data storage structure (Abstract data type).
- Unlike arrays, access of elements in a stack is restricted.
- It has two main functions
    - push
    - pop
- Insertion in a stack is done using push function and removal from a stack is done using pop function.
- Stack allows access to only the last element inserted hence, an item can be inserted or removed from the stack from one end called the top of the stack. It is therefore, also called Last-In-First-Out (LIFO) list.
- Stack has three properties:
    - *capacity* stands for the maximum number of elements stack can hold
    - *size* stands for the current size of the stack
    - *elements* is the array of elements.

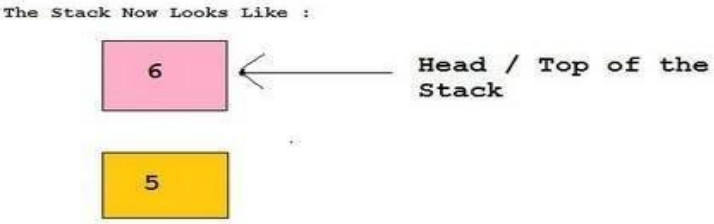
The Stack: Last In-First Out (LIFO)

The Empty Stack:
(null)

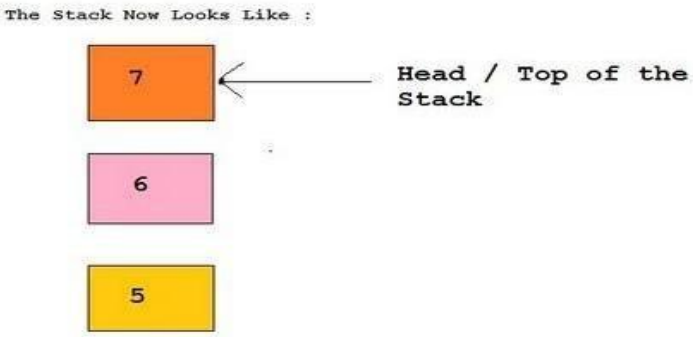
Push 5 onto the Stack:



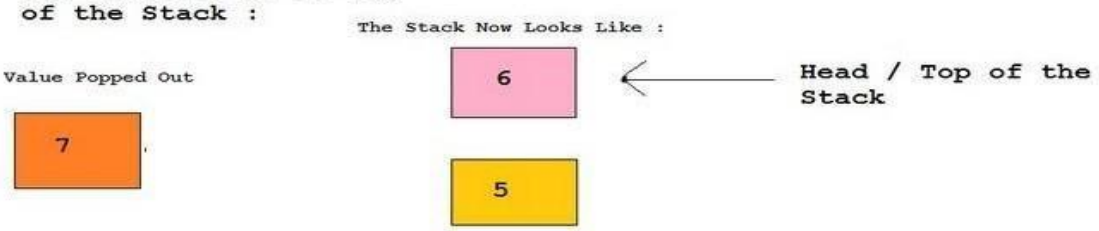
Push 6 onto the Stack:



Push 7 onto the Stack:



Pop Whatever is on top of the Stack :



Pop Whatever is on top of the Stack :

1. **createStack function**– This function takes the maximum number of elements (maxElements) the stack can hold as an argument, creates a stack according to it and returns a pointer to the stack. It initializes Stack S using malloc function and its properties.
2. **push function** - This function takes the pointer to the top of the stack S and the item (element) to be inserted as arguments. Check for the emptiness of stack
3. **pop function** - This function takes the pointer to the top of the stack S as an argument.
4. **top function** – This function takes the pointer to the top of the stack S as an argument and returns the topmost element of the stack S.

Properties of stacks:

1. Each function runs in O(1) time.
2. It has two basic implementations
   - Array-based implementation – It is simple and efficient but the maximum size of the stack is fixed.
   - Singly Linked List-based implementation – It's complicated but there is no limit on the stack size, it is subjected to the available memory.

Stacks - C Program source code

```
#include<stdio.h>
#include<stdlib.h>
/* Stack has three properties. capacity stands for the maximum number of elements stack can hold.
   Size stands for the current size of the stack and elements is the array of elements */
typedef struct Stack
{
      int capacity;
      int size;
      int *elements;
}Stack;
/* crateStack function takes argument the maximum number of elements the stack can hold, creates
   a stack according to it and returns a pointer to the stack. */
Stack * createStack(int maxElements)
{
      /* Create a Stack */
      Stack *S;
      S = (Stack *)malloc(sizeof(Stack));
      /* Initialise its properties */
      S->elements = (int *)malloc(sizeof(int)*maxElements);
      S->size = 0;
      S->capacity = maxElements;
      /* Return the pointer */
      return S;
}
void pop(Stack *S)
{
```

```
        /* If stack size is zero then it is empty. So we cannot pop */
        if(S->size==0)
        {
                printf("Stack is Empty\n");
                return;
        }
        /* Removing an element is equivalent to reducing its size by one */
        else
        {
                S->size--;
        }
        return;
}
int top(Stack *S)
{
        if(S->size==0)
        {
                printf("Stack is Empty\n");
                exit(0);
        }
        /* Return the topmost element */
        return S->elements[S->size-1];
}
void push(Stack *S,int element)
{
        /* If the stack is full, we cannot push an element into it as there is no space for it.*/
        if(S->size == S->capacity)
        {
                printf("Stack is Full\n");
        }
        else
        {
                /* Push an element on the top of it and increase its size by one*/
                S->elements[S->size++] = element;
        }
```

```
        return;
}
int main()
{
        Stack *S = createStack(5);
        push(S,7);
        push(S,5);
        push(S,21);
        push(S,-1);
        printf("Top element is %d\n",top(S));
        pop(S);
        printf("Top element is %d\n",top(S));
        pop(S);
        printf("Top element is %d\n",top(S));
        pop(S);
        printf("Top element is %d\n",top(S));

}
```

Evaluation of Arithmetic Expressions

Stacks are useful in evaluation of arithmetic expressions. Consider the expression
5 * 3 +2 + 6 * 4
The expression can be evaluated by first multiplying 5 and 3, storing the result in A, adding 2 and A, saving the result in A. We then multiply 6 and 4 and save the answer in B. We finish off by adding A and B and leaving the final answer in A.
A = 15 2 +
= 17
B = 6 4 *
= 24
A = 17 24 +
= 41
We can write this sequence of operations as follows:
5 3 * 2 + 6 4 * +

This notation is known as postfix notation and is evaluated as described above. We shall shortly show how this form can be generated using a stack.
Basically there are 3 types of notations for expressions. The standard form is known as the infix form. The other two are postfix and prefix forms.

Infix: operator is between operands A + B
Postfix : operator follows operands A B +
Prefix: operator precedes operands + A B

Note that all infix expressions can not be evaluated by using the left to right order of the operators inside the expression. However, the operators in a postfix expression are ALWAYS in the correct evaluation order. Thus evaluation of an infix expression is done in two steps.

The first step is to convert it into its equivalent postfix expression. The second step involves

evaluation of the postfix expression. We shall see in this section, how stacks are useful in carrying out both the steps. Let us first examine the basic process of infix to postfix conversion.

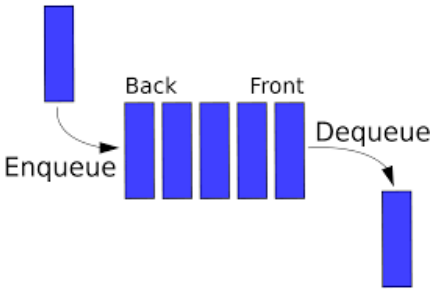
Infix to postfix conversion:

a + b * c Infix form
(precedence of * is higher than of +)
a + (b * c) convert the multiplication
a + (b c *) convert the addition
a (b c *) + Remove parentheses
a b c * + Postfix form
Note that there is no need of parentheses in postfix forms.

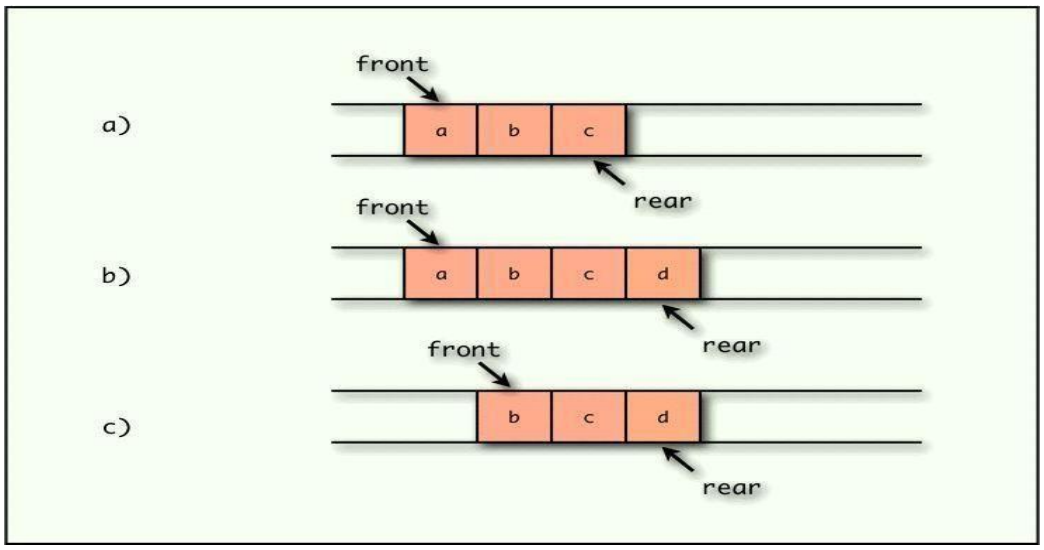
Example 2:
(A + B) * C Infix form
(A B +) * C Convert the addition
(A B +) C * Convert multiplication
A B + C * Postfix form
No need of parenthesis anywhere
Example 3:
a + ((b * c) / d) 5
a + ((b c *) /d)
(precedence of * and / are same and they are left associative)
a + (b c * d /)
a b c * d / +

Queue ADT

- It is a linear data structure that maintains a list of elements such that insertion happens at rear end and deletion happens at front end.
- FIFO – First In First Out principle



Logical Representation of queues:



Queue implementation:

- Array implementation of queues
- Linked list implementation of queues

Array Implementation of queues:

Insert operation(Enqueue)

- It includes checking whether or not the queue pointer rear is pointing at the upper bound of the array. If it is not, rear is incremented by 1 and the new item is placed at the end of the queue.
- **Algorithm**

  insert(queue[max],element, front ,rear)

  Step 1 : Start

  Step 2 : If front = NULL goto Step 3 else goto Step 6

  Step 3 : front = rear = 0

  Step 4 : queue[front] = element

  Step 5 : Goto Step 10

  Step 6 : If rear = MAX-1 goto Step 7 else goto Step 8

  Step 7 : Display the message,"Queue is FULL" and goto Step 10

  Step 8 : rear = rear + 1

  Step 9 : queue[rear] = element

  Step 10 : Stop

- **Delete operation(Dequeue)**
  - It includes checking whether or not the queue pointer front is already pointing at NULL. . If it is not, the item that is being currently pointed is removed from the queue and front pointer is incremented by 1.
  - Algorithm

delete(queue[MAX], front , rear)

Step 1 : Start

Step 2 : If front = NULL and rear =NULL goto Step 3 else goto Step 4

Step 3 : Display the message,"Queue is Empty" and goto Step 10

Step 4 : If front != NULL and front = rear goto Step 5 else goto Step 8

Step 5 : Set I = queue[front]

Step 6 : Set front = rear = -1

Step 7 : Return the deleted element I and goto Step 10

Step 8 : Set i=queue[front]

Step 9 : Return the deleted element i

Step 10 : Stop

Program to implement queues using arrays:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int queue[100];
int front=-1;
int rear=-1;
void insert(int);
int del();
void display();
void main()
{
        int choice;
        int num1=0,num2=0;
while(1)
```

```
{
        printf("\n Select a choice from the following : ");
        printf("\n[1] Add an element into the queue");
        printf("\n[2] Remove an element from the queue");
        printf("\n[3] Display the queue elements");
        printf("\n[4] Exit\n");
        scanf("%d",&choice);
        switch(choice)
        {
        case 1:
        {
        printf("\nEnter the element to be added :");
        scanf("%d",&num1);
        insert(num1); break;
        }
        {
        num2=del();
         if(num2==9999);
        else
        printf("\n %d element removed from the queue");
        getch(); break;
        }
case 3:
         {
                display();
                getch();
                break;
                }
                Case 4:
                Exit(1);
```

```
                    break;
                default:
                    printf("\nInvalid choice");
                    break;
            }
        }
}
void display()
{
        int i;
        if(front==-1)
        {
            printf("Queue is empty");
            return;
        }
        printf("\n The queue elements are :\n");
        for(i=front;i<=rear;i++)
            printf("\t%d",queue[i]);
}
void insert(int element)
{
        if(front==-1 )
        {
            front = rear = front +1;
            queue[front] = element;
            return;
        }
        if(rear==99)
        {
            printf("Queue is full");
            getch();
```

```
            return;
        }
        rear = rear +1;
        queue[rear]=element;
}
void insert(int element)
{
        if(front==-1 )
        {
            front = rear = front +1;
            queue[front] = element;
            return;
        }
        if(rear==99)
        {
            printf("Queue is full");
            getch();
            return;
        }
        rear = rear +1;
        queue[rear]=element;
}
```

Linked Implementation of Queues:

- It is based on the dynamic memory management techniques which allow allocation and de-allocation of memory space at runtime.

Insert operation:

It involves the following subtasks :

1. Reserving memory space of the size of a queue element in memory
2. Storing the added value at the new location
3. Linking the new element with existing queue
4. Updating the *rear* pointer

insert(structure queue, value, front , rear)

Step 1: Start

Step 2: Set ptr = (struct queue*)malloc(sizeof(struct queue))

Step 3 : Set ptr→element=value

Step 4 : if front =NULL goto Step 5 else goto Step 7

Step 5 : Set front =rear = ptr

Step 6 : Set ptr→next = NULL and goto Step 10

Step 7 : Set rear→next = ptr

Step 8 : Set ptr→next =NULL

Step 9 : Set rear = ptr

Step 10 : Stop

Delete operation:

It involves the following subtasks :

1. Checking whether queue is empty
2. Retrieving the front most element of the queue
3. Updating the front pointer
4. Returning the retrieved value

delete(structure queue, front , rear)

Step 1 : Start

Step 2 : if front =NULL goto Step 3 else goto Step 4

Step 3 : Display message, “Queue is empty” and goto Step 7

Step 4 : Set i = front→element

Step 5 : Set front = front→next

Step 6 : Return the deleted element i

Step 7 : Stop

Program to implement queues using linked lists:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
```

```
struct queue
{
        int element;
        struct queue *next;
};
struct queue *front = NULL;
struct queue *rear = NULL;
void insert(int);
int del();
void display();
void main()
{
        int choice;
        int num1=0,num2=0;
        while(1)
        {
        printf("\n Select a choice from the following : ");
        printf("\n[1] Add an element into the queue");
        printf("\n[2] Remove an element from the queue");
        printf("\n[3] Display the queue elements");
         printf("\n[4] Exit\n");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
 printf("\nEnter the element to be added :");
 scanf("%d",&num1);
 insert(num1); break;
 }
 case 2:
 {
 num2=del();
```

```
 if(num2==9999)

;
else
printf("\n %d element removed from the queue");
getch();

 break;

}
case 3:
{
display();

getch();

break;

}
case 4:
exit(1);

break;

default:

printf("\nInvalid choice");

break;

                }

        }

}

void insert(int element)

{

        struct queue * ptr = (struct queue*)malloc(sizeof(struct queue));

        ptr→element=value;

        if(front =NULL)

        {
front =rear = ptr;

ptr→next = NULL;
 }

else

{
```

```
                rear→next = ptr;
                ptr→next =NULL;
                rear = ptr;
        }
    }
    int del()
    {
        int i;
        if(front == NULL) /*checking whether the queue is empty*/
        {
return(-9999);
    }
    else
    {
        i = front→element;
        front = front→next;
        return i;
    }
    }
    void display()
    {
        struct queue *ptr = front;
        if(front==NULL)
        {
            printf("Queue is empty");
            return;
        }
```