DATA STRUCTURE PROJECT

1.ASGAR ALI **ROLL NO 06SI4003**

MATHEMATICS DEPARTMENT

2.SHANTANU BOSE **ROLL NO 06SI4004**

MATHEMATICS DEPARTMENT

3.JAIDEV **ROLL NO 06PH6204**

PHYSICS DEPARTMENT

4.K.SAMBASIVA RAO **ROLL NO 06PH6206**

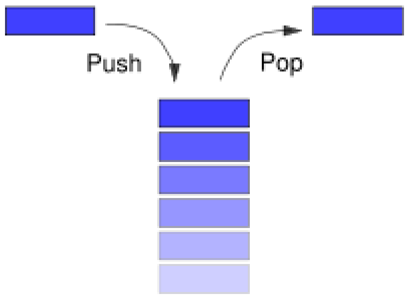
PHYSICS DEPARTMENT

THE STACK

A *stack* is an ordered collection of items into which new items may be inserted and items may be deleted at one end, called the *top* of the stack.

Primitive Operations

The two changes that can be made to a stack are given special names. When an element is added to a stack, it is *pushed* and when an item is removed it is *popped* from the stack. Because of the push operation which adds an element to a stack, a stack is sometimes called a *pushdown list.*



There is no upper limit on the number of items that may be kept in a stack, that is the *push* operation is applicable to any stack. But one cannot *pop* an element from an *empty stack* (stack containing no elements). So checking whether the stack is empty is very much necessary in the *pop* operation.

Complexity

A typical storage requirement for a stack of *n* elements is *O(n)*. The typical time requirement of O(1) operations is also easy to satisfy with a dynamic array or (singly) linked list implementation

C Implementation of Stack

Done by Asgar Ali(06SI4003)

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

```
typedef struct
{
	int top;
	int items[200];
}stack;
int empty(stack *ps)
{	if(ps->top==-1)
		return(1);
	else
		return(0);
}
void popandtest(stack *ps,int *px,int *pund)
{
	if(empty(ps))
	{	*pund=1;
		return;
	}
	*pund=0;
	*px=ps->items[ps->top--];
	return;

}
void push(stack *ps,int x)
{
	ps->items[++ps->top]=x;
	return;
}
void main()
{
	int x,choice,i,und;
	stack s;
	FILE *fp;
	fp=fopen("stk.txt","w");
	s.top=-1;
	fprintf(fp,"pop :2   push :1     exit :4");
	fprintf(fp,"Choice?");
	scanf("%d",&choice);
	fprintf(fp,"%d\n",choice);
	while(choice!=4)
	{	if(choice==1)
		{	fprintf(fp,"Give the element to be pushed");
			scanf("%d\n",&x);
			fprintf(fp,"%d",x);
			push(&s,x);
		}
		else
```

```
                {       popandtest(&s,&x,&und);
                        if(und)
                                fprintf(fp,"Underflow.Push elements first.\n");
                        else
                                fprintf(fp,"Popped element=%d\n",x);
                }
                fprintf("stack:    ");
                for(i=0;i<=s.top;++i)
                        fprintf(fp,"%d   ",s.items[i]);
                fprintf(fp,"\n");
                fprintf(fp,"Choice?");
                scanf("%d",&choice);
                fprintf(fp,"%d\n",choice);
        }
        fprintf(fp,"You have chosen to terminate the application");
}
```

OUTPUT

```
pop :2   push :1     exit :4
Choice?1
Give the element to be pushed 23
stack:    23
Choice?1
Give the element to be pushed 45
stack:    23   45
Choice?1
Give the element to be pushed 56
stack:    23   45   56
Choice?2
Popped element=56
stack:    23   45
Choice?2
Popped element=45
stack:    23
Choice?2
Popped element=23

Choice?2
Underflow.Push elements first.

Choice?4
You have chosen to terminate the application
```

Applications of Stack

Let us consider the sum of a and b.We think of applying the *operator* '+' to the *operands* a and b and write the sum as a+b
This particular representation is called infix. There are two alternative notations for expressing the sum of a and b using the symbols a,b,+. These are

ab+	**postfix**
+ab	**prefix**

These are important topics of computer science in it's own right. Given any one of them, we can express it in the other two forms, or evaluate them.
These can be done magnificently by using stack.

A. Evaluation of a postfix operation

Each operator in a postfix string refers to the previous two operands of the string. Suppose that each time we read an operand we push it onto a stack. When we reach an operator, its operands will be the top two elements of the stack. We can then pop these two elements, perform the indicated operation on them, and push the result on the stack so that it will be available for use as an operand of the next operator. The algorithm is as follows

```
opndstk=the empty stack
while(not the end of input)
    symb=next input character
    if(symb is an operand)
            push(opndstk,symb)
    else
            opnd2=pop(opndstk)
            opnd1=pop(opndstk)
            value=result of applying symb to opnd1 and opnd2
            push(opndstack,value)

return(pop(opndstk))
```

Let us now consider the example
6 2 3 + - 3 8 2 / + * 2 ^ 3 +

The algorithm proceeds as follows

symb	opnd1	opnd2	value	opndstk
6				6
2				6,2

3				**6,2,3**
+	**2**	**3**	**5**	**6,5**
-	**6**	**5**	**1**	**1**
3	**6**	**5**	**1**	**1,3**
8	**6**	**5**	**1**	**1,3,8**
2	**6**	**5**	**1**	**1,3,8,2**
/	**8**	**2**	**4**	**1,3,4**
+	**3**	**4**	**7**	**1,7**
*****	**1**	**7**	**7**	**7**
2	**1**	**7**	**7**	**7,2**
^	**7**	**2**	**49**	**49**
3	**7**	**2**	**49**	**49,3**
+	**49**	**3**	**52**	**52**

Value=52

A. Changing an expression into equivalent forms

From Infix to postfix

The algorithm is given as follows

```
opstk= the empty stack
while( not the end of input)
    symb=next input character
    if( symb is an operand)
            add symb to the postfix string
    else
            while( !empty( opstk) && prcd( stacktop( opstk),symb))
/*here the prcd(a,b) is a function which returns TRUE or FALSE according as' a' is
of higher or lower precedenge than 'b'   */
                    topsymb= pop( opstk)
                    add topsymb to the postfix string
            if( empty(opstk)  ||  symb != ')')
                    push(opstk, symb)
            else
                    topsymb= pop( opstk)
while( !empty(opstk))
    topsymb= pop(opstk)
    add topsymb to the postfix string
```