UNIT-III

Stacks: Introduction to stacks: properties and operations, implementing stacks using arrays and linked lists, Applications of stacks in expression evaluation, backtracking, reversing list etc.

1.Introduction to stacks

What is a Stack?

A stack is a **linear data structure** where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted.

A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages.

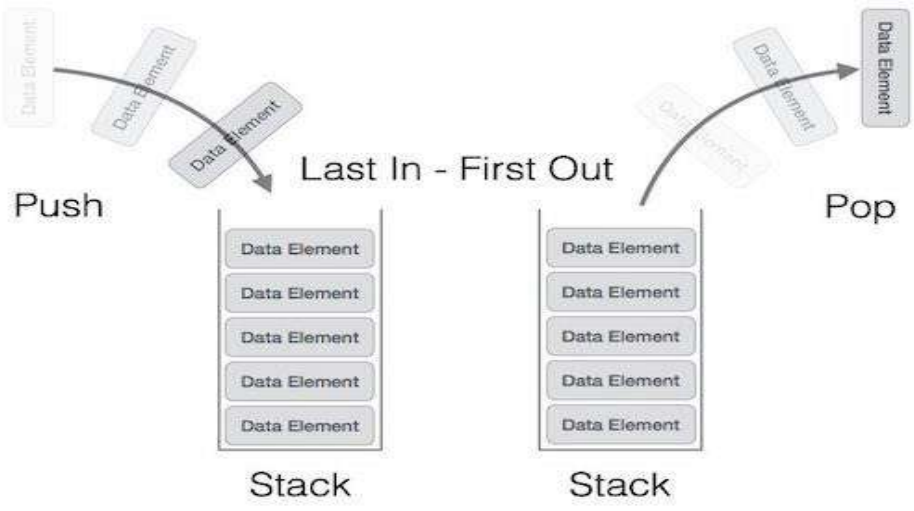
It is named stack because it has the similar operations as the real-world stacks, for example − a pack of cards or a pile of plates, etc.

Stack is considered a complex data structure because it uses other data structures for implementation, such as Arrays, Linked lists, etc.

Stack Representation

A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack. so the last element added to the stack will be the first element removed from the stack.

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

1.1 Properties of stack

LIFO:

The fundamental property of a stack is that the last element added is the first one to be removed.

Linear data structure:

Stack follows a linear data structure, where elements are arranged in a sequence , and operations are occur at one end, known as top

Limited access:

Elements in a stack can only be accessed at top

Dynamic in size:

Stacks can grow or shrinks as elements are added or removed

Efficient operations:

The basic operations on stack, such as pushing and popping elements are typically very efficient.

1.2 Operations on a stack

The most fundamental operations in the stack ADT include:

push(), pop(), peek(), isFull(), isEmpty().

1.Stack Insertion: push()

Push: Push operation is used to add new elements in to the stack.

At the time of addition first check the stack is full or not.

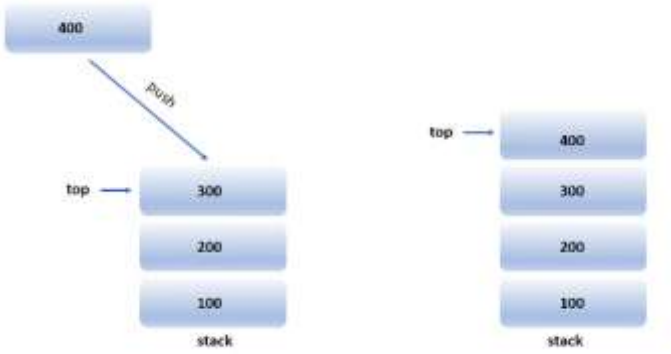
If the stack is full it generates an error message "stack overflow".

The push() is an operation that inserts elements into the stack.

The following is an algorithm that describes the push() operation in a simpler way.

Algorithm

1. Checks if the stack is full.
2. If the stack is full, produces an error and exit.
3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.



2.**Stack Deletion: pop()**

Pop: Pop operation is used to delete elements from the stack.

At the time of deletion firstcheck the stack is empty or not.

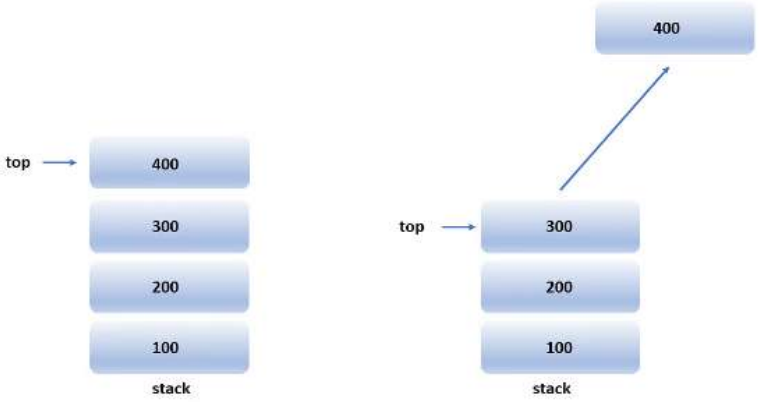
If the stack is empty it generates an error message "stack underflow".

The *pop()* is a data manipulation operation which removes elements from the stack.

The following pseudo code describes the pop() operation in a simpler way.

Algorithm

1. Checks if the stack is empty.
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.
5. Returns success.



3.Retrieving topmost Element from Stack: peek()

The *peek()* is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.

4.Verifying whether the Stack is full: isFull()

The *isFull()* operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the size of the stack is equal to the top position of the stack, the stack is full.

   Return 1.
3. Otherwise, return 0.
4. END

5.Verifying whether the Stack is empty: isEmpty()

The *isEmpty()* operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the top value is -1, the stack is empty. Return 1.
3. Otherwise, return 0.
4. END

2. Implementing stack using Arrays

Let us consider a stack with 5 elements capacity. This is called as the size of the stack.

The number of elements to be added should not exceed the maximum size of the stack.

If we attempt to add new element beyond the maximum size, we will encounter a **stack overflow** condition.

Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a **stack underflow** condition.

Operations

Implementing stack using arrays are having following operations

1.Insertion (push)

2.Deletion (pop)

3.Display

1.push(): When an element is added to a stack, the operation is performed by push().

Algorithm: Procedure for push():

Step 1: START

Step 2: if top>=size-1 then Write " Stack is Overflow"

Step 3: Otherwise

3.1: read data value 'x'

3.2: top=top+1;

3.3: stack[top]=x;

Step 4: END

Below Figure shows the creation of a stack and addition of elements using push().

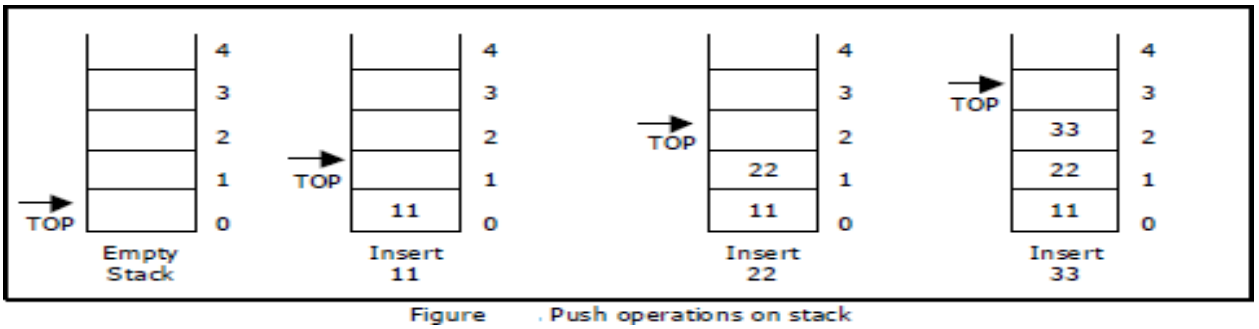


Figure . Push operations on stack

Initially **top=-1**, we can insert an element in to the stack, increment the top value i.e **top=top+1**.

We can insert an element in to the stack first check the condition is stack is full or not. i.e **top>=size-1**. Otherwise add the element in to the stack.

2.Pop(): When an element is taken off from the stack, the operation is performed by pop().

Algorithm: procedure pop():

Step 1: START
Step 2: if top==-1 then
Write "Stack is Underflow"
Step 3: otherwise
3.1: print "deleted element"
3.2: top=top-1;
Step 4: END

Below figure shows a stack initially with three elements and shows the deletion of elements using pop().

Decrement the top value i.e **top=top-1**.

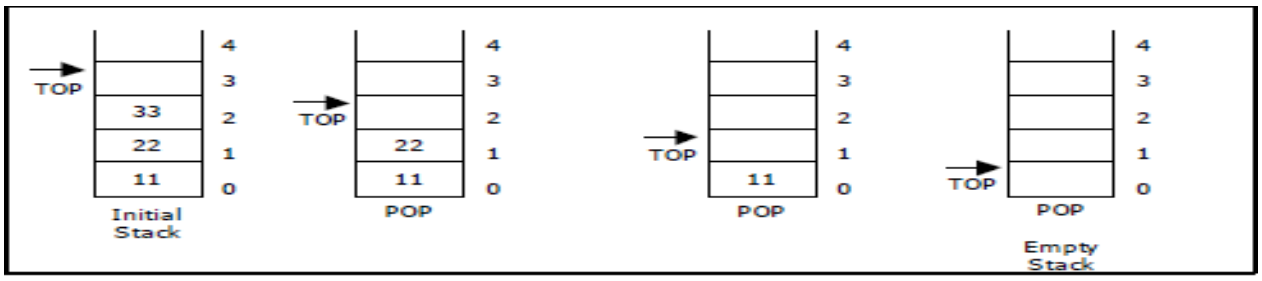


Figure Pop operations on stack

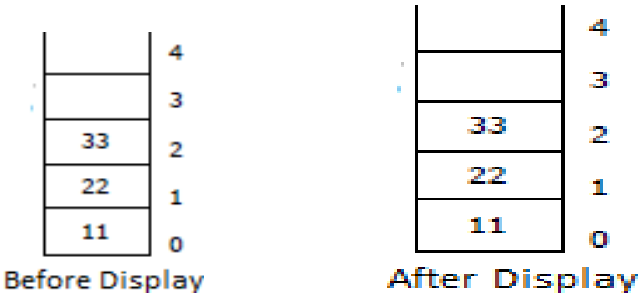
We can delete an element from the stack first check the condition is stack is empty or not.
i.e **top==-1**. Otherwise remove the element from the stack.

3.display(): This operation performed display the elements in the stack. We display the element in the stack check the condition is stack is empty or not i.e top==-1.Otherwise display the list of elements in the stack.

Algorithm: procedure pop():

Step 1: START
Step 2: if top==-1 then
Write "Stack is Underflow"
Step 3: otherwise
3.1: print "Display elements are"
3.2: for top to 0
Print 'stack[i]'

Step 4: END



STACK USING ARRAYS

```
#include<stdio.h>

#include<conio.h>

int stack[50],top=-1,n,i,item;

void push();

void pop();

void display();

void main()

{

        int ch;

        clrscr();

        printf("Enter the size of the array:");

        scanf("%d",&n);
```

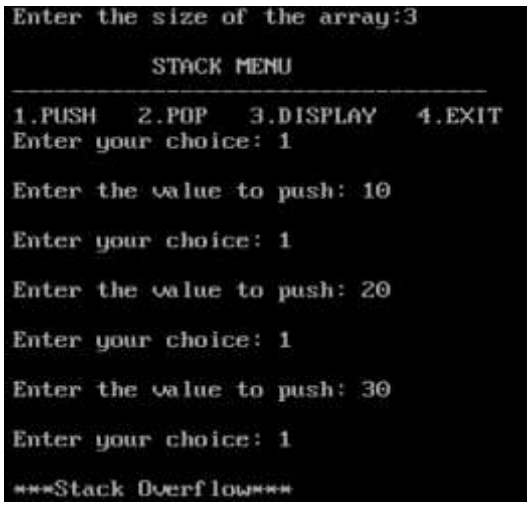
```
        printf("\n          STACK MENU");
        printf("\n---------------------------------");
        printf("\n1.PUSH   2.POP   3.DISPLAY   4.EXIT");
        while(1)
        {
                printf("\nEnter your choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: push();
                                break;
                        case 2: pop();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(1);
                                break;
                }
        }
        getch();
}
void push()
{
        if(top==n-1)
        {
                printf("\n***Stack Overflow***");
        }
        else
```

```
        {
                printf("\nEnter the value to push: ");
                scanf("%d",&item);
                top++;
                stack[top]=item;

        }
}
void pop()
{
        if(top==-1)
        {
                printf("\n***Stack Underflow***");
        }
        else
        {
                printf("\nPopped item is %d",stack[top]);
                top--;
        }
}
void display()
{
        if( top == -1)
        {
                printf("\nThere are no elements in the stack to print.");
        }
        else
        {
```

```
            printf("\nElements are:");
            for(i=0;i<=top;i++)
            {
                printf("%d ",stack[i]);
            }
        }
}
```

OUTPUT:



3.Stack Implementation using Linked List:

Linked 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.

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 following Fig 7.13

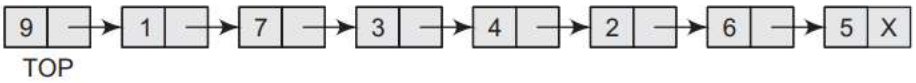


Figure 7.13 Linked stack

Representation of linked stack

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer. The linked stack looks as shown in figure.

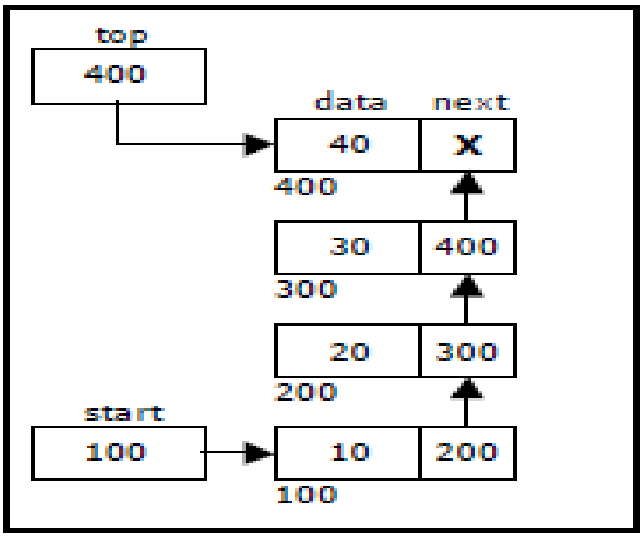


Figure Linked stack representation

OPERATIONS ON A LINKED STACK

1.Push Operation:

The push operation is used to insert an element into the stack. The new element is added at the top most 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.

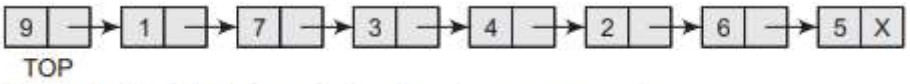


Figure 7.15 Linked stack after inserting a new node

Algorithm

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

Step 4: END

2.Pop Operation

The pop operation is used to delete the topmost element from a stack.

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.

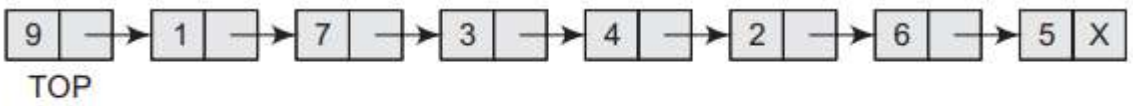


Figure 7.17 Linked stack

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.

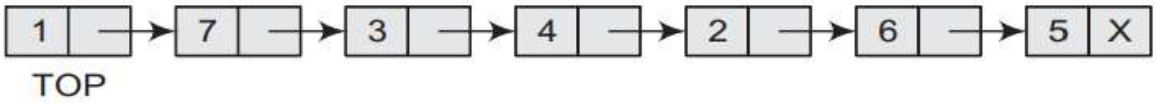


Figure 7.18 Linked stack after deletion of the topmost element

Algorithm

Step 1: IF TOP = NULL

PRINT UNDERFLOW [END OF IF]

Goto Step 5

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP-> NEXT

Step 4: FREE PTR

Step 5: END

Stack using linked list

```
#include<stdlib.h>
#include<stdio.h>
struct node
{
        int data;
        struct node *next;
};
struct node *top,*temp,*new_node;
void push();
void pop();
void display();
void search();
void main()
{
        int choice;
        clrscr();

        while(1)
        {
```

```
            printf("\n          MAIN MENU          ");
            printf("\n1.PUSH 2.POP 3.DISPLAY 4.Exit\n");
            printf("*****************************************");
            printf("\nEnter your choice:");
            scanf("%d",&choice);
            switch(choice)
            {
                  case 1: push();
                        break;
                  case 2: pop();
                        break;
                  case 3: display();
                        break;
                  case 4: exit(1);
                        break;
                  defaulf:printf("Invalid option");
                        break;
            }
      }

}
 void push()
{
      new_node = (struct node*)malloc(sizeof(struct node));
      printf("Enter the data:");
      scanf("%d",&new_node->data);
      new_node->next=NULL;
      if(top==NULL)
```

```
	{
		top=new_node;
	}
	else
	{
		new_node->next = top;
		top = new_node;
	}
	getch();
}
void pop()
{
	if(top==NULL)
	{
		printf("\nThere are no elements in the List.");
	}
	else	if(top->next==NULL)
		{
			printf("\nDeleted node is %d", top->data);
			top=NULL;
			free(top);
		}
		else
		{	printf("\nDeleted node is: %d ", top->data);
			top=top->next;

		}
	getch();
```

```
}
void display()
{

        if(top==NULL)
        {
                printf("STACK UNDERFLOE / There are no elements in the STACK");
        }
        else
        {       printf("\nElements in the STACK are:\n");
                temp = top;
                while(temp != NULL)
                {
                        printf("%d\n",temp->data);
                        temp=temp->next;
                }
        }
}
```

Applications of stack:

- 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
- Tower of Hanoi

4.Evaluating Algebraic expressions:

An **algebraic expression** is a legal combination of operators and operands.
Operand is the quantity on which a mathematical operation is performed.
Operand may be a variable like x,y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands.
Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations.
They are infix,postfix and prefix notations:
Infix: It is the form of an arithmetic expression in which we fix (place) the arithmeticoperator in between the two operands.
Example: A + B
Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmeticoperator before (pre) its two operands. The prefix notation is called as polish notation.
Example: + A B
Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as suffix notation and is also referred to reverse polish notation.

Example: A B +

4.1. Evaluation of postfix expression:
The postfix expression is evaluated easily by the use of a stack.

1. When a number is seen, it is pushed onto the stack;
2. When an operator is seen, the operator is applied to the two numbers that arepopped from the stack and the result is pushed onto the stack.
3. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
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

4.2 Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
   a) If the scanned symbol is an operand, then place directly in the postfix expression (output).
   b) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
   c) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

   The three important features of postfix expression are:
2. The operands maintain the same order as in the equivalent infix expression.
3. The parentheses are not needed to designate the expression unambiguously.
4. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Example 1:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

5.Back Tracking

A backtracking algorithm is a problem-solving algorithm that uses a **brute force approach** for finding the desired output.

The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.

The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.

Example-QUEENS PROBLEM:

This 8 queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queensattack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

- The solution vector X (X1…Xn) represents a solution in which Xi is the column of the th rowwhere I th queen is placed.
- First, we have to check no two queens are in same row.
- Second, we have to check no two queens are in same column.
- The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I th queen, where I represents the row and X (j) represents the column position.
- Third, we have to check no two queens are in it diagonal.
- Consider two dimensional array A[1:n,1:n] in which we observe that every element on thesame diagonal that runs from upper left to lower right has the same value.
- Also, every element on the same diagonal that runs from lower right to upper left has the samevalue.
- Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|I-k|.

STEPS TO GENERATE THE SOLUTION:

❖ Initialize x array to zero and start by placing the first queen in k=1 in the first row.

❖ To find the column position start from value 1 to n, where 'n' is the no. Of columns or no. Ofqueens.

❖ If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Here we have to checkwhether there is any queen in the same column or diagonal.

❖ For this considers the previous position, which had already, been found out. Check whetherX (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.

❖ If any one of the conditions is true then return false indicating that k th queen can't be placed inposition X (k).

❖ For not possible condition increment X (k) value by one and precede d until the position isfound.

- If the position X (k) □ n and k=n then the solution is generated completely.
- If k<n, then increment the ‘k’ value and find position of the next queen.
- If the position X (k)>n then k th queen cannot be placed as the size of the matrix is ‘N*N’.
- So decrement the ‘k’ value by one i.e. we have to back track and after the position of the previous queen.

Algorithm Nqueen (k,n)

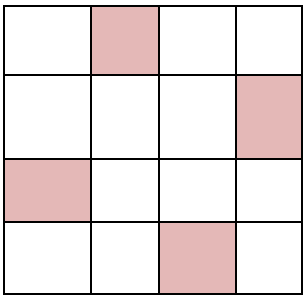
```
//using backtracking it prints all possible positions of n queens in ‘n*n’ chessboard. So
//that they are non-tracking.
{
For I=1 to n do
{
If place (k,I) then
{
X [k]=I;
If (k=n)
then write (X [1:n]);
Else
nquenns(k+1,n)                    ;
}
}
}
```

Example: 4 queens.

Two possible solutions are

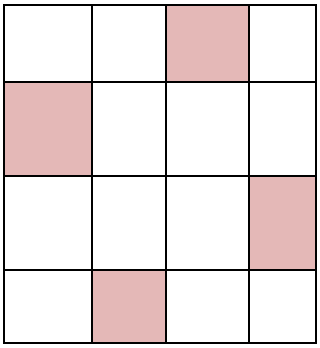
Solutin-1

(2 4 1 3)



Solution 2

(3 1 4 2)



6.Reversing List

How to Reverse a Linked List using Stack

A linked list is a data structure that consists of a sequence of nodes, where each node contains data and a reference (or link) to the next node in the sequence. The last node in the list points to None, indicating the end of the list.

Reversing a linked list means changing the order of the nodes so that the last node becomes the first node, the second-to-last node becomes the second node, and so on, resulting in a completely reversed sequence.

One approach to reversing a linked list is by using a stack. A stack is a Last-In-First-Out (LIFO) data structure, meaning that the last element added to the stack is the first one to be removed.

Input:

Output:

Algorithm on How to Reverse a Linked List using Stack.

- Check if the linked list is empty or has a single node. If yes, then no need to reverse it and simply return from the function. Otherwise, follow the below steps.
- Declare an empty stack.
- Iterate through the linked list and push the values of the nodes into the stack.
- Now, after the iteration is complete, the linked list is stored in reverse order in the stack.
- Now, start popping the elements from the stack and iterating through the linked list together, and change the values in the nodes by the values we get from the stack.
- Once the stack is empty, we will have the required reversed linked list.