# Unit – III

**Syllabus:**

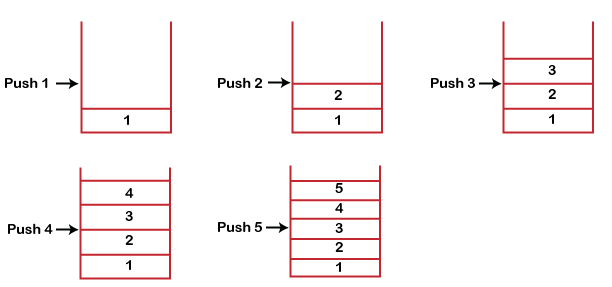
* Stack: representation
* Stack implementation using arrays and linked lists
* Applications of stack

**INTRODUCTION:**

* A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**).
* It contains only one pointer **top pointer** pointing to the topmost element of the stack.
* Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
* **A stack is defined as a container in which insertion and deletion happen at only one end called the top of the stack.**
* It is called as stack because it behaves like a real-world stack, piles of books, etc.
* A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
* It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

**WORKING OF STACK:**

* Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.
* Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



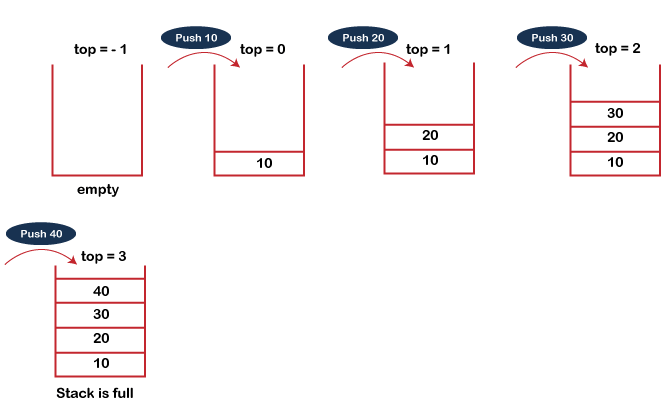
* Since our stack is full as the size of the stack is 5.
* In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack.
* The stack gets filled up from the bottom to the top.
* When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed.
* It follows the LIFO pattern, which means that the value entered first will be removed last.
* In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

**ADT OF STACK:**

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [push](https://www.javatpoint.com/insertion-in-singly-linked-list-at-beginning) | Insert an element at top of the stack |
| 2 | [pop](https://www.javatpoint.com/insertion-in-singly-linked-list-at-end) | Delete an element from top of a stack |
| 3 | Peek | Return the top element of the stack without deleting it from stack |
| 4 | [count](https://www.javatpoint.com/deletion-in-singly-linked-list-at-beginning) | Count the number of elements in the stack |
| 5 | [change](https://www.javatpoint.com/deletion-in-singly-linked-list-at-end) | Change a particular element in the stack |
| 6 | display | To display elements in the stack |
| 7 | [is\_Full](https://www.javatpoint.com/traversing-in-singly-linked-list) | To check whether a stack is full or not(Returns a Boolean value) |
| 8 | Is\_Empty | To check whether a stack is empty or not(Returns a Boolean value) |

# Push Operation

* Before inserting an element in a stack, we check whether the stack is full.
* If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
* When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
* When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
* The elements will be inserted until we reach the ***max*** size of the stack.



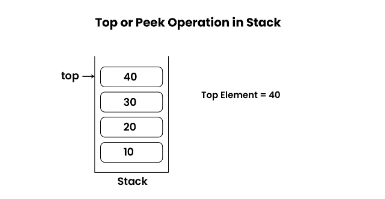
# Pop Operation

* Before deleting the element from the stack, we check whether the stack is empty.
* If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
* If the stack is not empty, we first access the element which is pointed by the ***top***
* Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.

# 

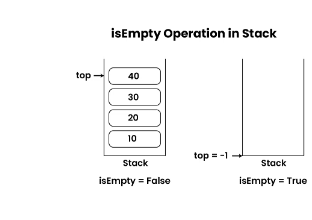
**Peek Operation**

* Before returning the top element from the stack, we check if the stack is empty.
* If the stack is empty (top == -1), we simply print “Stack is empty”.
* Otherwise, we return the element stored at **index = top**.



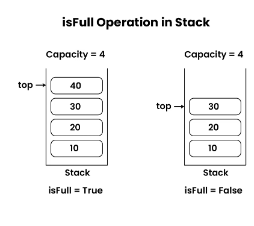
**is\_Empty Operation**

* Check for the value of **top**in stack.
* If **(top == -1)**, then the stack is **empty**so return **true**.
* Otherwise, the stack is not empty so return **false**.



**is\_Full Operation**

* Check for the value of **top**in stack.
* If **(top == MAX\_SIZE-1),**then the stack is **full**so return **true**.
* Otherwise, the stack is not full so return **false**.



**Code for implementing stack using arrays**

#include <stdio.h>

#define MAX\_SIZE 10

void push(int);

int pop(void);

int peek(void);

int is\_empty();

int is\_full();

void display();

int count();

void change(int pos,int ele);

int stack[MAX\_SIZE];

int top=-1,pos,ele,data;

int main()

{

char userContinue = 'Y';

int userChoice;

while (userContinue == 'Y' || userContinue == 'y') {

printf("\n------ STACK DATA STRUCTURE ------\n\n");

printf("\n1. Push the data");

printf("\n2. Pop the data");

printf("\n3. Peek the data");

printf("\n4. Stack Empty");

printf("\n5. Stack Full");

printf("\n6. Display Stack");

printf("\n7. Count Stack Elements");

printf("\n8. Change element at given position");

printf("\n9. Exit");

printf("\n\nEnter Your Choice: ");

scanf("%d", &userChoice);

switch (userChoice)

{

case 1:

printf("\n\nEnter Data: ");

scanf("%d", &ele);

push(ele);

break;

case 2:

data = pop();

if(data!=0)

printf("\nValue %d was popped", data);

break;

case 3:

data = peek();

if(data!=0)

printf("\nValue %d was peeked", data);

break;

case 4:

if(is\_empty()==0){

display();

}

else{

printf("Stack is empty\n");

}

break;

case 5:

if(is\_full()==0){

printf("Stack is empty\n");

}

else{

display();

}

break;

case 6:

display();

break;

case 7:

int l=count();

if(l==0){

printf("Stack is empty\n");

}

else{

printf("The number of elements in the stack are %d\n",l);

}

break;

case 8:

printf("Enter the position at which element to be inserted:");

scanf("%d",&pos);

printf("\nEnter the element to be inserted:");

scanf("%d",&ele);

change(pos,ele);

break;

default:

printf("\n\tInvalid Choice!");

}

printf("\n\nDo you want to continue? ");

fflush(stdin);

scanf(" %c", &userContinue);

}

return 0;

}

int is\_full()

{

if(top==MAX\_SIZE-1){

return 1;

}

return 0;

}

int is\_empty()

{

if(top==-1){

return 1;

}

return 0;

}

void push(int data)

{

if(top==MAX\_SIZE-1){

printf("Stack Overflow\n");

return;

}

stack[++top]=data;

}

int pop()

{

if(top==-1){

printf("Stack is empty\n");

return 0;

}

return stack[top--];

}

int peek()

{

if(top==-1){

printf("Stack is empty\n");

return 0;

}

return stack[top];

}

void display()

{

int i;

if(top==-1){

printf("Stack is empty\n");

return;

}

for(i=top;i>=0;i--){

printf("%d ",stack[i]);

}

}

int count(){

if(top==-1){

printf("Stack is empty\n");

return 0;

}

return top+1;

}

void change(int pos, int ele){

if (pos<0||pos>top){

printf("\nCannot insert at given postion");

return;

}

stack[pos]=ele;

}

**Code to implement Stack using Linked Lists**

#include <stdio.h>

#include<stdlib.h>

#define MAX\_SIZE 10

void push(int);

int pop(void);

int peek(void);

int is\_empty();

int is\_full();

void display();

int count1();

void change(int pos,int ele);

struct stack{

int data;

struct stack \*next;

};

struct stack \*top=NULL;

int pos,ele,data;

static int count=0;

int main()

{

char userContinue = 'Y';

int userChoice;

while (userContinue == 'Y' || userContinue == 'y') {

printf("\n------ STACK DATA STRUCTURE ------\n\n");

printf("\n1. Push the data");

printf("\n2. Pop the data");

printf("\n3. Peek the data");

printf("\n4. Stack Empty");

printf("\n5. Stack Full");

printf("\n6. Display Stack");

printf("\n7. Count Stack Elements");

printf("\n8. Change element at given position");

printf("\n9. Exit");

printf("\n\nEnter Your Choice: ");

scanf("%d", &userChoice);

switch (userChoice)

{

case 1:

printf("\n\nEnter Data: ");

scanf("%d", &ele);

push(ele);

break;

case 2:

data = pop();

if(data!=0)

printf("\nValue %d was popped", data);

break;

case 3:

data = peek();

if(data!=0)

printf("\nValue %d was peeked", data);

break;

case 4:

if(is\_empty()==0){

display();

}

else{

printf("Stack is empty\n");

}

break;

case 5:

if(is\_full()==0){

printf("Stack is empty\n");

}

else{

display();

}

break;

case 6:

display();

break;

case 7:

int l=count1();

if(l==0){

printf("Stack is empty\n");

}

else{

printf("The number of elements in the stack are %d\n",l);

}

break;

case 8:

printf("Enter the position at which element to be inserted:");

scanf("%d",&pos);

printf("\nEnter the element to be inserted:");

scanf("%d",&ele);

change(pos,ele);

break;

default:

printf("\n\tInvalid Choice!");

}

printf("\n\nDo you want to continue? ");

fflush(stdin);

scanf(" %c", &userContinue);

}

return 0;

}

int is\_full()

{

if(count==MAX\_SIZE){

return 1;

}

else if(top==NULL && count==0){

return 0;

}

else{

return count;

}

}

int is\_empty()

{

if(count==0){

return 1;

}

return 0;

}

void push(int data)

{

if(count==MAX\_SIZE-1){

printf("Stack Overflow\n");

return;

}

struct stack\* newNode = (struct stack\*)malloc(sizeof(struct stack));

newNode->data = data;

newNode->next = top;

top = newNode;

count++;

}

int pop()

{

if(top==NULL && count==0){

printf("Stack is empty\n");

return 0;

}

struct stack \*temp = top;

int x=temp->data;

top = top->next;

free(temp);

count--;

return x;

}

int peek()

{

if(top==NULL && count==0){

printf("Stack is empty\n");

return 0;

}

return top->data;

}

void display()

{

int i;

if(top== NULL && count==0){

printf("Stack is empty\n");

return;

}

struct stack \*temp=top;

while(temp!=NULL){

printf("%d ",temp->data);

temp=temp->next;

}

}

int count1(){

if(top == NULL && count==0){

printf("Stack is empty\n");

return 0;

}

return count;

}

void change(int pos, int ele){

if (pos<0||pos>count){

printf("\nCannot insert at given postion");

return;

}

struct stack\* temp = top;

for (int i = 1; i < pos; i++) {

temp = temp->next;

}

temp->data = ele;

display();

}

**APPLICATIONS OF STACK:**

* Expression Evaluation
* Checking parenthesis are balanced or not
* Recursion implementation
* Backtracking implementation
* Reversing list
* Memory Management

**EXPRESSION REPRESENTATION:**

## There are 3 different ways to represent expressions

## Infix Notation

## Prefix Notation

## Postfix Notation

## Infix Expressions

**Infix expressions** are mathematical expressions where the **operator is placed between its operands**. This is the most common mathematical notation used by humans. For example, the expression “2 + 3” is an infix expression, where the operator “+” is placed between the operands “2” and “3”.

**Infix notation** is easy to read and understand for humans, but it can be difficult for computers to evaluate efficiently. This is because the order of operations must be taken into account, and parentheses can be used to override the default order of operations.

## Prefix Expressions (Polish Notation)

**Prefix expressions**are also known as **Polish notation**, are a mathematical notation where the operator precedes its operands. This differs from the more common **infix notation**, where the operator is placed between its operand**s.**

In prefix notation, the operator is written first, followed by its operands. For example, the infix expression “a + b” would be written as “+ a b” in prefix notation.

## Postfix Expressions (Reverse Polish Notation)

**Postfix expressions**are also known as **Reverse Polish Notation (RPN)**, are a mathematical notation where the**operator follows its operands**. This differs from the more common infix notation, where the operator is placed between its operands.

In postfix notation, operands are written first, followed by the operator. For example, the infix expression “5 + 2” would be written as “5 2 +” in postfix notation.

**INFIX EVALUATION:**

Iterate through given expression, one character at a time

1. If the character is an operand, push it to the operand stack.
2. If the character is an operator,
   1. If the operator stack is empty then push it to the operator stack.
   2. Else If the operator stack is not empty,
      * If the character's precedence is greater than or equal to the precedence of the stack top of the operator stack, then push the character to the operator stack.
      * If the character's precedence is less than the precedence of the stack top of the operator stack then do Process (as explained above) until character's precedence is less or stack is not empty.
3. If the character is "(", then push it onto the operator stack.
4. If the character is ")", then do ***Process*** (as explained above) until the corresponding "(" is encountered in operator stack. Now just pop out the "(".

Once the expression iteration is completed and the operator stack is not empty, do ***Process*** until the operator stack is empty.  The values left in the operand stack is our final result.

**Code for evaluating infix expression**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

int precedence(char c) {

switch (c) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

}

return -1;

}

int performOperation(int \*operands, int \*top\_operand, char \*operations, int \*top\_operation) {

int a = operands[(\*top\_operand)--];

int b = operands[(\*top\_operand)--];

char operation = operations[(\*top\_operation)--];

switch (operation) {

case '+':

return b + a;

case '-':

return b - a;

case '\*':

return a \* b;

case '/':

if (a == 0) {

printf("Cannot divide by zero\n");

return 0;

}

return b / a;

}

return 0;

}

int isOperator(char c) {

return (c == '+' || c == '-' || c == '/' || c == '\*' || c == '^');

}

int evaluate(char \*exp) {

int operands[100];

int top\_operand = -1;

char operations[100];

int top\_operation = -1;

for (int i = 0; exp[i]; i++) {

char c = exp[i];

if (isdigit(c)) {

int num = 0;

while (isdigit(c)) {

num = num \* 10 + (c - '0');

i++;

if (exp[i]) {

c = exp[i];

} else {

break;

}

}

i--;

operands[++top\_operand] = num;

} else if (c == '(') {

operations[++top\_operation] = c;

} else if (c == ')') {

while (operations[top\_operation] != '(') {

int output = performOperation(operands, &top\_operand, operations, &top\_operation);

operands[++top\_operand] = output;

}

top\_operation--;

} else if (isOperator(c)) {

while (top\_operation >= 0 && precedence(c) <= precedence(operations[top\_operation])) {

int output = performOperation(operands, &top\_operand, operations, &top\_operation);

operands[++top\_operand] = output;

}

operations[++top\_operation] = c;

}

}

while (top\_operation >= 0) {

int output = performOperation(operands, &top\_operand, operations, &top\_operation);

operands[++top\_operand] = output;

}

return operands[top\_operand];

}

int main() {

char infixExpression[100];

printf("Enter infix expression:");

scanf("%s",infixExpression);

printf("%d\n", evaluate(infixExpression));

return 0;

}

**POSTFIX EVALUATION**

**POSTFIX & PREFIX EVALUATION:**

* Create a stack to store operands (or values).
* Scan the given expression from left to right and do the following for every scanned element.
  + If the element is a number, push it into the stack.
  + If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
* When the expression is ended, the number in the stack is the final answer.

**Code for Evaluating Postfix Expression**

#include<stdio.h>

int top = -1, stack [100];

void push (int);

int pop( );

int eval (char, int, int);

main ( ){

char a[50], ch;

int i,op1,op2,res,x;

printf("enter a postfix expression: \n");

gets (a);

for(i=0; a[i]!='\0'; i++){

ch = a[i];

if (ch>='0' && ch<='9')

push(ch-'0');

else{

op2 = pop ( );

op1 = pop ( );

res = eval (ch, op1, op2);

push (res);

}

}

x = pop ( );

printf("evaluated value = %d", x);

}

void push (int n){

top++;

stack [top] = n;

}

int pop ( ){

int res ;

res = stack [top];

top--;

return res;

}

int eval (char ch, int op1, int op2){

switch (ch){

case '+' : return (op1+op2);

case '-' : return (op1-op2);

case '\*' : return (op1\*op2);

case '/' : return (op1/op2);

}

}

**PREFIX EVALUATION**

* Create a stack to store operands (or values).
* Scan the given expression from right to left and do the following for every scanned element.
  + If the element is a number, push it into the stack.
  + If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
* When the expression is ended, the number in the stack is the final answer.

**Code to evaluate prefix expression**

#include<stdio.h>

#include<string.h>

int top = -1, stack [100];

main ( ){

char a[50], ch;

int i,op1,op2,res,x;

void push (int);

int pop( );

int eval (char, int, int);

printf("enter a postfix expression: \n");

gets (a);

for(i=strlen(a)-1; i>=0; i--){

ch = a[i];

if (ch>='0' && ch<='9')

push(ch-'0');

else{

op2 = pop ( );

op1 = pop ( );

res = eval (ch, op1, op2);

push (res);

}

}

x = pop ( );

printf("evaluated value = %d", x);

}

void push (int n){

top++;

stack [top] = n;

}

int pop ( ){

int res ;

res = stack [top];

top--;

return res;

}

int eval (char ch, int op1, int op2){

switch (ch){

case '+' : return (op1+op2);

case '-' : return (op1-op2);

case '\*' : return (op1\*op2);

case '/' : return (op1/op2);

}

}

**INFIX TO POSTFIX CONVERSION:**

* Scan the infix expression **from left to right**.
* If the scanned character is an operand, put it in the postfix expression.
* Otherwise, do the following
  + If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a ‘**(**‘ ], then push it in the stack. [‘**^**‘ operator is right associative and other operators like ‘**+**‘,’**–**‘,’**\***‘ and ‘**/**‘ are left-associative].
    - Check especially for a condition when the operator at the top of the stack and the scanned operator both are ‘**^**‘. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
    - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
  + Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
    - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
* If the scanned character is a ‘**(**‘, push it to the stack.
* If the scanned character is a ‘**)**’, pop the stack and output it until a ‘**(**‘ is encountered, and discard both the parenthesis.
* Repeat steps **2-5** until the infix expression is scanned.
* Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
* Finally, print the postfix expression.

**Code for Infix to Postfix Conversion**

#include <stdio.h>

#include<ctype.h>

#define max 100

char s[max];

int top = -1;

void push(char element) {

s[++top]=element;

}

char pop(){

return(s[top--]);

}

int priority(char element){

switch (element)

{

case '(':

return 1;

case '+':

case '-':

return 2;

case '\*':

case '/':

return 3;

}

}

int main(void) {

char infix[50],postfix[50],ch,element;

int i=0,k=0;

printf("-------------------READ THE INFIX-------------------------\n\n");

scanf("%s",infix);

while((ch = infix[i++])!='\0'){

if(ch=='(')

push(ch);

else if(isalnum(ch))

postfix[k++]=ch;

else if(ch==')'){

while(s[top]!='(')

{

postfix[k++]=pop();

}

element=pop();

}

else{

while(priority(s[top])>=priority(ch))

postfix[k++]=pop();

push(ch);

}

}

while(top!=-1){

postfix[k++]=pop();

}

postfix[k]='\0';

printf("\n\nGiven Infix Expn: %s Postfix Expn: %s\n", infix, postfix);

return 0;

}

**INFIX TO PREFIX CONVERSION:**

* ***Step 1:****Reverse the infix expression. Note while reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.*
* ***Step 2:****Convert the reversed*infix expression to “nearly” postfix expression*.*
  + *While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.*
* ***Step 3:****Reverse the postfix expression.*

**Code to convert infix to prefix expression**

#include<stdio.h>

#define max 100

int top=-1, a[max];

void push(char x)

{

a[++top]=x;

}

char pop()

{ if(top==-1)

return -1;

else

return a[top--];

}

int prcd(char c)

{ if(c==')')

return 0;

else if(c=='+'||c=='-')

return 1;

else if(c=='\*'||c=='/')

return 2;

}

void infixtoprefix(char infix[max],char prefix[max])

{

char temp,x;

int i=0,j=0;

strrev(infix);

while(infix[i]!='\0')

{

temp=infix[i];

if(isalnum(temp))

{

prefix[j++]=temp;

}

else if(temp==')')

push(temp);

else if(temp=='(')

{

while((x=pop())!=')')

{

prefix[j++]=x;

}

}

else

{ while(prcd(a[top])>=prcd(temp))

{prefix[j++]=pop();}

push(temp);

}

i++;

}

while(top!= -1)

prefix[j++]=pop();

prefix[j]='\0';

strrev(prefix);

}

int main()

{

char infix[max],prefix[max];

printf("Enter the infix expression\n");

gets(infix);

printf("The infix expression is %s\n",infix);

infixtoprefix(infix, prefix);

printf("The prefix expression is %s\n",prefix);

return 0;

}

**BALANCED PARANTHESES:**

* Initialize an empty stack.
* Iterate through the string.
* If the current character is an opening bracket, push it onto the stack.
* If the current character is a closing bracket, pop the top element off the stack and compare it to the current character.
* If the two characters do not match, the parentheses are not balanced.
* If the stack is empty at the end of the string, the parentheses are balanced.

**Code for checking balanced paranthesis**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 100

char stack[MAX\_SIZE];

int top = -1;

void push(char data) {

if (top == MAX\_SIZE - 1) {

printf("Overflow stack!\n");

return;

}

top++;

stack[top] = data;

}

char pop() {

if (top == -1) {

printf("Empty stack!\n");

return ' ';

}

char data = stack[top];

top--;

return data;

}

int is\_Matching(char char1, char char2) {

if (char1 == '(' && char2 == ')') {

return 1;

} else if (char1 == '[' && char2 == ']') {

return 1;

} else if (char1 == '{' && char2 == '}') {

return 1;

} else {

return 0;

}

}

int is\_Balanced(char\* text) {

int i;

for (i = 0; i < strlen(text); i++) {

if (text[i] == '(' || text[i] == '[' || text[i] == '{') {

push(text[i]);

} else if (text[i] == ')' || text[i] == ']' || text[i] == '}') {

if (top == -1) {

return 0;

} else if (!is\_Matching(pop(), text[i])) {

return 0;

}

}

}

if (top == -1) {

return 1;

} else {

return 0;

}

}

int main() {

char text[MAX\_SIZE];

printf("Input an expression in parentheses: ");

scanf("%s", text);

if (is\_Balanced(text)) {

printf("The expression is balanced.\n");

} else {

printf("The expression is not balanced.\n");

}

return 0;

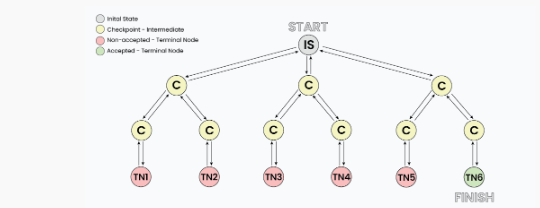
}

**BACKTRACKING:**

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying **different options** and **undoing** them if they lead to a **dead end**. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

**Working of BackTracking**

* *As shown in the image, “****IS”****represents the****Initial State****where the recursion call starts to find a valid solution.*
* ***C :****it represents different****Checkpoints****for recursive calls*
* ***TN****: it represents the****Terminal Nodes****where no further recursive calls can be made, these nodes act as base case of recursion and we determine whether the current solution is valid or not at this state.*
* *At each Checkpoint, our program makes some decisions and move to other checkpoints untill it reaches a terminal Node, after determining whether a solution is valid or not, the program starts to revert back to the checkpoints and try to explore other paths. For example in the above image****TN1…TN5****are the terminal node where the solution is not acceptable, while****TN6****is the state where we found a valid solution.*
* *The back arrows in the images shows backtracking in actions, where we revert the changes made by some checkpoint.*



**Program to implement BackTracking using Stack**

**Given a string print all permutations of the string**

**Example:**

**ABC**

**Output**

**ABC**

**ACB**

**BAC**

**BCA**

**CBA**

**CAB**

* Create a function **permute()** with parameters as input string, starting index of the string, ending index of the string
* Call this function with values input string, 0, size of string – 1
  + In this function, if the value of  L and R is the same then print the same string
    - * Else run a for loop from L to R and swap the current element in the for loop with the inputString[L]
      * Then again call this same function by increasing the value of L by 1
      * After that again swap the previously swapped values to initiate backtracking

#include <stdio.h>

#include <string.h>

void swap(char\* x, char\* y)

{

    char temp;

    temp = \*x;

    \*x = \*y;

    \*y = temp;

}

 void permute(char\* a, int l, int r)

{

    int i;

    if (l == r)

        printf("%s\n", a);

    else {

        for (i = l; i <= r; i++) {

            swap((a + l), (a + i));

            permute(a, l + 1, r);

            swap((a + l), (a + i)); // backtrack step

        }

    }

}

int main()

{

    char str[100];

printf(“Enter a string:”);

scanf(“%s”,s);

    int n = strlen(str);

    permute(str, 0, n - 1);

    return 0;

}