LECTURE NOTES ON

DATA STRUCTURES

B.Tech III Sem (IARE-R18)

By

Dr. P Govardhan, Associate Professor
Mrs. S J Sowjanya, Assistant Professor

DEPARTMENT OF CIVIL ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD - 500 043

Module – I

Introduction to Data Structures, Searching and Sorting

Introduction:

What is data structure?

"The way information is organized in the memory of a computer is called a **data structure"**.

(OR)

A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

Definition of data structures

- Many algorithms require that we use a proper representation of data to achieve efficiency.
- This representation and the operations that are allowed for it are called data structures.
- Each data structure allows insertion, access, deletion etc.

Why do we need data structures?

- Data structures allow us to achieve an important goal: component reuse
- Once each data structure has been implemented once, it can be used over and over again in various applications.

Common data structures are

Stacks	Queues	Lists
Trees	Graphs	Tables

Classification of Data Structure:

Based on how the data items or operated it will classified into

1. **Primitive Data Structure** is one the data items are operated closest to the machine level instruction.

Eg : int, char and double.

2. **Non-Primitive Data Structure** is one that data items are not operated closest to machine level instruction.

Linear Data Structure : In which the data items are stored in sequence order.

Eg: Arrays, Lists, Stacks and Queues.

Operations performed on any linear structure:

1. Traversal – Processing each element in the list
2. Search – Finding the location of the element with a given value.
3. Insertion – Adding a new element to the list.
4. Deletion – Removing an element from the list.
5. Sorting – Arranging the elements in some type of order.
6. Merging – Combining two lists into a single list.

Non Linear Data Structure: In which the order of data items is not presence.

Eg : Trees, Graphs.

Searching Techniques:

Searching is one of the most common problems that arise in computing. Searching is the algorithmic process of finding a particular item in a collection of items. A search typically answers either True or False as to whether the item is present. On occasion it may be modified to return where the item is found. Search operations are usually carried out on a key field. Well, to search an element in a given array, there are two popular algorithms available:

1. Linear Search

2. Binary Search

Linear Search:

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found. It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1. Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.

2. It has a time complexity of O(n), which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.

Source Code:

```
def linear_search(obj, item, start=0):
    for i in range(start, len(obj)):
        if obj[i] == item:
            return i
  return -1
arr=[1,2,3,4,5,6,7,8]
x=4
```

```
result=linear_search(arr,x)
if result==-1:
   print ("element does not exist")
else:
   print ("element exist in position %d" %result)
```

Binary Search:

1. Binary Search is used with sorted array or list. In binary search, we follow the following steps:

2. We start by comparing the element to be searched with the element in the middle of the list/array.

3. If we get a match, we return the index of the middle element.

4. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.

5. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step1.

If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1. Binary Search is useful when there is large number of elements in an array and they are sorted. So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

1. It is great to search through large sorted arrays.

2. It has a time complexity of O(log n) which is a very good time complexity

Source code:

```
array =[1,2,3,4,5,6,7,8,9]

def binary_search(searchfor,array):
   lowerbound=0
   upperbound=len(array)-1
   found=False
   while found==False and lowerbound<=upperbound:
      midpoint=(lowerbound+upperbound)//2
      if array[midpoint]==searchfor:
         found =True
         return found
      elif array[midpoint]<searchfor:
         lowerbound=midpoint+1
      else:
         upperbound=midpoint-1
   return found
```

```
searchfor=int(input("what are you searching for?"))
if binary_search(searchfor,array):
   print ("element found")
else:
   print ("element not found")
```

Operations on Data Structures:

The basic operations that are performed on data structures are as follows:

Insertion: Insertion means addition of a new data element in a data structure.

Deletion: Deletion means removal of a data element from a data structure if it is found.

Searching: Searching involves searching for the specified data element in a data structure.

Traversal: Traversal of a data structure means processing all the data elements present in it.

Sorting: Arranging data elements of a data structure in a specified order is called sorting.

Merging: Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

Sorting Techniques:

Sorting is the basic operation in computer science. Sorting is the process of arranging data in some given sequence or order (in increasing or decreasing order).

For example you have an array which contains 10 elements as follow:

10, 3 ,6 12, 4, 17, 5, 9

After sorting value must be:

3, 4, 5, 6, 9, 10, 12, 17

Bubble sort:

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:
First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Source Code:

```
# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])
```

Selection sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Source Code:

```
# Python program for implementation of Selection
# Sort
import sys
A = [64, 25, 12, 22, 11]
 # Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with the first element

    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i])
```

Insertion sort:

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the

value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Source Code:

```
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key


# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])
```

Comparison of Sorting Algorithms:

In this table, n is the number of records to be sorted. The columns "Best" ,"Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. The run times and the memory requirements listed below should be understood to be inside big O notation.Time Complexity comparison of Sorting Algorithms:

Algorithm	**Data Structure**	**Time Complexity**		
		Best	**Average**	**Worst**
Quick sort	Array	O(n log(n))	O(n log(n))	O(n^2)
Merge sort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)

Space Complexity comparison of Sorting Algorithms:

Algorithm	**Data Structure**	**Worst Case Auxiliary Space Complexity**
Quick sort	Array	O(n)
Merge sort	Array	O(n)
Bubble Sort	Array	O(1)
Insertion Sort	Array	O(1)
Select Sort	Array	O(1)
Bucket Sort	Array	O(nk)

Module – II

Linear Data Structures

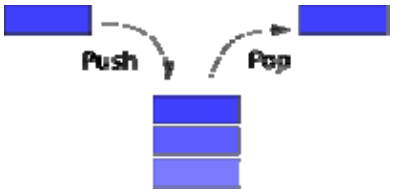
Stacks:

Primitive operations

"A stack is an ordered list in which all insertions and deletions are made at one end, called the top". Stacks are sometimes referred to as Last In First Out (LIFO) lists

Stacks have some useful terminology associated with them:

- **Push** To add an element to the stack
- **Pop** To remove an element from the stock
- **Peek** To look at elements in the stack without removing them
- **LIFO** Refers to the last in, first out behavior of the stack
- **FILO** Equivalent to LIFO



Simple representation of a stack

Given a stack S=(a[1],a[2],.....a[n]) then we say that a1 is the bottom most element and element a[i]) is on top of element a[i-1], 1<i<=n.

Implementation of stacks using arrays

1. array (static memory).
2. linked list (dynamic memory)

The operations of stack is

1. PUSH operations
2. POP operations
3. PEEK operations

The Stack ADT

A stack S is an abstract data type (ADT) supporting the following three methods:

push(n): Inserts the item **n** at the top of stack

pop() : Removes the top element from the stack and returns that top element. An error occurs if the stack is empty.

peek() : Returns the top element and an error occurs if the stack is empty.

1.Adding an element into a stack. (called PUSH operations)

Adding element into the TOP of the stack is called PUSH operation.

Check conditions :

TOP = N , then **STACK FULL**

where N is maximum size of the stack.

PUSH algorithm**procedure** add(item : items);

{

add item to the global stack stack ; top is the current top of stack and n is its maximum size}

begin

if top = n **then** stackfull; top := top+1;

stack(top) := item;

end: {of add}

Implementation in C using array:

/* here, the variables stack, top and size are global variables */

void push (int item)

{

if (top == size-1)

printf("Stack is Overflow");

else

{

top = top + 1;

stack[top] = item;

}

2.Deleting an element from a stack. (called POP operations)

Deleting or Removing element from the TOP of the stack is called POP operations.

Check Condition:

TOP = 0**, then** STACK EMPTY

Deletion in stack (POP Operation)

procedure delete(**var** item : items);

{

remove top element from the stack stack and put it in the item}

begin

if top = 0 **then** stackempty; item := stack(top);

top := top-1;

end; {of delete}

3. Peek Operation:

- Returns the item at the top of the stack but does not delete it.
- This can also result in **underflow** if the stack is empty.

Source Code:

```
# Function to create a stack. It initializes size of stack as 0
def createStack():
    stack = []
    return stack

# Stack is empty when stack size is 0
def isEmpty(stack):
    return len(stack) == 0

# Function to add an item to stack. It increases size by 1
def push(stack, item):
    if(len(stack)==size):
        print("overflow")
        return
    stack.append(item)

# Function to remove an item from stack. It decreases size by 1
def pop(stack):
    if (isEmpty(stack)):
        print("underflow")
        return
    return stack.pop()

#Function to know peek element
```

```
def peek(stack):
    if(isEmpty(stack)):
        print("stack empty")
        return
    else:
        n=len(stack)
        print("peek element is: ",stack[n-1])

#Function to display stack
def display(stack):
    print(stack)

# Driver program to test above functions
stack = createStack()
size=int(input("enter the size of stack"))

print("Menu\n1.push(p)\n2.pop(o)\n3.peek(e)")

choice=1
while choice!='q':
    print("enter your choice")
    ch=input()
    choice=ch.lower()
    if choice=='p':
        push(stack,int(input("enter a value")))
        display(stack)
    elif choice=='o':
        pop(stack)
        display(stack)
    elif choice=='e':
        peek(stack)
    else:
        print("enter proper choice")
```

Applications of stacks arithmetic

1. It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
2. Infix to Postfix Transformation
3. It is useful during the execution of recursive programs
4. A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
5. A stack (memory stack) can be used in function calls including recursion.
6. Reversing Data
7. Reverse a List

8. Convert Decimal to Binary

9. Parsing – It is a logic that breaks into independent pieces for further processing

10. Backtracking

Examples :

1. Infix notation A+(B*C) equivalent Postfix notation ABC*+

2. Infix notation (A+B)*C Equivalent Postfix notation AB+C*

Expression evaluation and syntax parsing

Calculators employing reverse Polish notation (also known as postfix notation)use a stack structure to hold values.

Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines. Note that natural languages are context sensitive languages and stacks alone are not enough to interpret their meaning.

Infix, Prefix and Postfix Notation:

We are accustomed to write arithmetic expressions with the operation between the two operands: a+b or c/d. If we write a+b*c, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
40 - 3 * 5 + 1		

Source Code:

```
# Python program to evaluate value of a postfix expression

# Class to convert the expression
```

```
class Evaluate:

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)


    # The main function that converts given infix expression
    # to postfix expression
    def evaluatePostfix(self, exp):

        # Iterate over the expression for conversion
        for i in exp:

            # If the scanned character is an operand
            # (number here) push it to the stack
            if i.isdigit():
                self.push(i)

            # If the scanned character is an operator,
            # pop two elements from stack and apply it.
            else:
                val1 = self.pop()
                val2 = self.pop()
                self.push(str(eval(val2 + i + val1)))

        return int(self.pop())

# Driver program to test above function
exp = "231*+9-"
obj = Evaluate(len(exp))
```

print ("Value of {0} is {1}".format(exp, obj.evaluatePostfix(exp)))

Application of stacks:

Arithmetic Expressions: Polish Notation

- An arithmetic expression will have operands and operators.
- Operator precedence listed below: Highest: ($)

Next Highest :(*) and (/)

Lowest :(+) and (-)

- For most common arithmetic operations, the operator symbol is placed in between its two operands. This is called infix notation.

Example: A + B , E * F

- Parentheses can be used to group the operations.

Example: (A + B) * C

- Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.
- Polish notation refers to the notation in which the operator symbol is placed before its two operands. This is called prefix notation.

Example: +AB, *EF

- The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.
- Accordingly, one never needs parentheses when writing expressions in Polish notation.
- Reverse Polish Notation refers to the analogous notation in which the operator symbol is placed after its two operands. This is called postfix notation.

Example: AB+, EF*

Here also the parentheses are not needed to determine the order of the operations.

The computer usually evaluates an arithmetic expression written in infix notation in two steps,

1. It converts the expression to postfix notation.
2. It evaluates the postfix expression.

In each step, the stack is the main tool that is used to accomplish the given task.

Algorithm postfix expression

Initialize a stack, opndstk to be empty.

{

scan the input string reading one element at a time into symb

}

While (not end of input string)

{

Symb := next input character;

If symb is an operand Then push (opndstk,symb)

Else

[symbol is an operator]

Opnd1:=pop(opndstk); Opnd2:=pop(opndnstk);

Value:=result of applying symb to opnd1 & opnd2 Push(opndstk,value);

Result := pop (opndstk);

Example:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Symbol	**Operand 1 (A)**	**Operand 2 (B)**	**Value (A Ä B)**	**STACK**
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	/	1, 3, 4
+	3	4	7	1, 7

*	1	7	7	7
2				7, 2
$	7	2	49	49
3				49, 3
+	49	3	52	**52**

The Final value in the STACK is 52. This is the answer for the given expression.

(1) run time stack for function calls (write factorial number calculation procedure)

push local data and return address onto stack

return by popping off local data and then popping off address and returning to it return value can be pushed onto stack before returning, popped off by caller

(2) expression parsing

e.g. matching brackets: [... (... (...) [...(...) ...] ...) ...]

push left ones, pop off and compare with right ones

4) Infix To Postfix Conversion

Infix expressions are often translated into postfix form in which the operators appear after their operands. **Steps:**

1. Initialize an empty stack.

2. Scan the Infix Expression from left to right.

3. If the scannned character is an operand, add it to the Postfix Expression.

4. If the scanned character is an operator and if the stack is empty, then push the character to stack.

5. If the scanned character is an operator and the stack is not empty, Then

(a) Compare the precedence of the character with the operator on the top of the stack.

(b) While operator at top of stack has higher precedence over the scanned character & stack is not empty.

(i) POP the stack.

(ii) Add the Popped character to Postfix String. (c) Push the scanned character to stack.

6. Repeat the steps 3-5 till all the characters

7. While stack is not empty,

(a) Add operator in top of stack

(b) Pop the stack.

8. Return the Postfix string.

Algorithm Infix to Postfix conversion (without parenthesis)

Opstk = the empty stack;

while (not end of input)

{

symb = next input character;

if (symb is an operand)

add symb to the Postfix String

else

{

While(! empty (opstk) && prec (stacktop (opstk), symb))

{

topsymb = pop (opstk)

add topsymb to the Postfix String;

} / * end of while */ Push(opstk, symb);

} /* end else */

} /* end while * /

While(! empty (opstk))

{

topsymb = pop (opstk)

add topsymb to the Postfix String

} /* end of while */

Return the Postfix String.

Source Code:

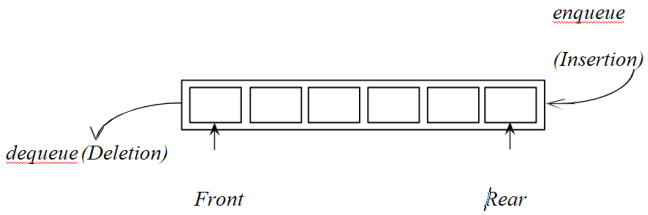
```
# Python program to convert infix expression to postfix

# Class to convert the expression
import string
```

```
class Conversion:

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []
        # Precedence setting
        self.output = []
        self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)

    # A utility function to check is the given character
    # is operand
    def isOperand(self, ch):
        return ch.isalpha()

    # Check if the precedence of operator is strictly
    # less than top of stack or not
    def notGreater(self, i):
        try:
            a = self.precedence[i]
            b = self.precedence[self.peek()]
            return True if a  <= b else False
        except KeyError:
            return False

    # The main function that converts given infix expression
    # to postfix expression
    def infixToPostfix(self, exp):

        # Iterate over the expression for conversion
        for i in exp:
            # If the character is an operand,
```

```
            # add it to output
            if self.isOperand(i):
                self.output.append(i)

            # If the character is an '(', push it to stack
            elif i  == '(':
                self.push(i)

            # If the scanned character is an ')', pop and
            # output from the stack until and '(' is found
            elif i == ')':
                while( (not self.isEmpty()) and self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

            # An operator is encountered
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)

        # pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())

        result= "".join(self.output)
        print(result)
# Driver program to test above function
exp = "a+b*(c^d-e)^(f+g*h)-i"
obj = Conversion(len(exp))
obj.infixToPostfix(exp)
```

QUEUE:

"A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT". Queues are sometimes referred to as First In First Out (FIFO)



Example

1. The people waiting in line at a bank cash counter form a queue.

2. In computer, the jobs waiting in line to use the processor for execution. This queue is called **Job Queue**.

Operations of Queue

There are two basic queue operations. They are,

Enqueue – Inserts an item / element at the rear end of the queue. An error occurs if the queue is full.

Dequeue – Removes an item / element from the front end of the queue, and returns it to the user. An error occurs if the queue is empty.

1. Addition into a queue

procedure addq (item : items);

{add item to the queue q}

begin

if rear=n **then** queuefull

else begin

rear :=rear+1; q[rear]:=item;

end; end;{of addq}

2. Deletion in a queue

procedure deleteq (**var** item : items);

{delete from the front of q and put into item}

begin

if front = rear **then** queueempty

else begin

front := front+1 item := q[front];

end; end

Source Code:

```
def enqueue(a,item):
    global r
    global f

    if r==-1 and f==-1:
        r=0
```

```
            f=0
            a.insert(r,item)

        elif r==(n-1):
            print("overflow")
            return
        else:
            r+=1
            a.insert(r,item)

        display(a)

def dequeue(a):
        global r
        global f
        if r==(n-1) and f==(n-1):
            item=a[f]
            r=-1
            f=-1
        elif r==-1 and f==-1:
            print("underflow")
            return
        else:
            item=a[f]
            f+=1
        print("deleted item is:",item)
        display(a)

def display(a):
        print("\ncurrent queue is:")
        for i in range(f,r+1):
            if f==-1 and r==-1:
                print("Queue is empty!")
                return
            print(a[i],end=" ")

#DC
n=int(input("enter the size of list"))
a=[]
r=-1
f=-1
print("Menu\n1.enqueue(e)\n2.dequeue(d)")
choice=1
while choice!='q':
        print("enter your choice")
        ch=input()
        choice=ch.lower()
        if choice=='e':
            enqueue(a,int(input("enter a value")))
            display(a)
        elif choice=='d':
            dequeue(a)
            display(a)
        else:
            print("enter proper choice")
```

Uses of Queues (Application of queue)

Queues remember things in first-in-first-out (FIFO) order. Good for fair (first come first served) ordering of actions.

Examples of use: (Application of stack)

1 scheduling

processing of GUI events printing request

2 simulation

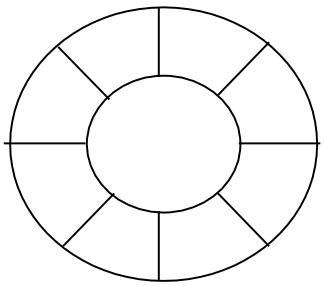
orders the events

models real life queues (e.g. supermarkets checkout, phone calls on hold)

Circular Queue :

Location of queue is viewed in a circular form. The first location is viewed after the last one.

Overflow occurs when all the locations are filled.



Algorithm Circular Queue Insert

```
Void CQInsert ( int queue[ ], front, rear, item)

{

if ( front = = 0 )

front = front +1;

if ( ( ( rear = maxsize ) && ( front = = 1 ) ) || ( ( rear ! = 0 ) && ( front = rear +1)))

{

printf( " queue overflow "); if( rear = = maxsize )
rear = 1;
else
rear = rear + 1; q [ rear ] = item;
}
}
```

Algorithm Circular Queue Delete

```
int CQDelete ( queue [ ], front, rear )

{

if ( front = = 0 )

printf ( " queue underflow ");

else

{

item = queue [ front ]; if(front = = rear )

{

front = 0; rear = 0;

}

else if ( front = = maxsize )

{

front = 1;

}

else

front = front + 1;

}

return item;

}
```

Double ended queue (deque):

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insertFront(): Adds an item at the front of Deque.
insertLast(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteLast(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported
getFront(): Gets the front item from queue.

getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.

Applications of Deque:

Since Deque supports both stack and queue operations, it can be used as both.

The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.

Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque.

Module – III

Linked List

Introduction to Linked List:

A **linked list** is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**. Each **node** is divided into two parts:

1. The first part contains the **information** of the element and

2. The second part contains the address of the next node (**link /next pointer field**) in the list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Linked lists using dynamic variables:

1. In array implementation of the linked lists a fixed set of nodes represented by an array is established at the beginning of the execution

2. A pointer to a node is represented by the relative position of the node within the array.

3. In array implementation, it is not possible to determine the number of nodes required for the linked list. Therefore;

a. Less number of nodes can be allocated which means that the program will have overflow problem.

b. More number of nodes can be allocated which means that some amount of the memory storage will be wasted.

4. The solution to this problem is to allow nodes that are **dynamic**, rather than static.

5. When a node is required storage is reserved /allocated for it and when a node is no longer needed, the memory storage is released /freed.

Advantages of linked lists:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.

2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.

3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.

4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.

2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

ARRAY	**LINKED LIST**
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during	It is not necessary to specify the number of elements during declaration (i.e., memory is
compile time).	allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

$P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$

2. Represent very large numbers and operations of the large number such as addition, multiplication and division.

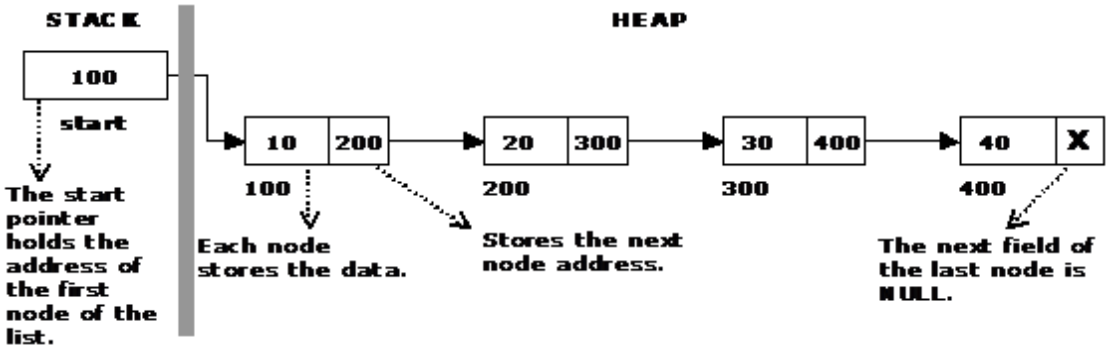
3. Linked lists are to implement stack, queue, trees and graphs.

4. Implement the symbol table in compiler construction.

Single Linked List:

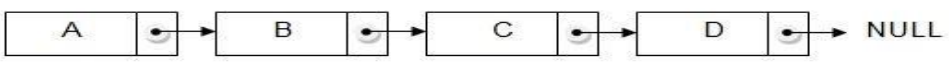
The simplest kind of linked list is a singly-linked list, which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

A singly linked list's node is divided into two parts. The first part holds or points to information about the node, and second part holds the address of next node. A singly linked list travels one way.



The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

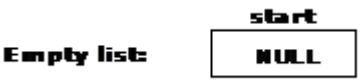
The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.



Single Linked List

Implementation of Single Linked List:

```
struct slinklist
{
        int data;
        struct slinklist* next;
};

typedef struct slinklist node;

node *start = NULL;
```



1. Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.Initialize the start pointer to be NULL.**The basic operations in a single linked list are:**

- Creation.
- Insertion.
- Deletion.
- Traversing.

Advantages of singly linked list:

1. Dynamic data structure.
2. We can perform deletion and insertion anywhere in the list.
3. We can merge two lists easily.

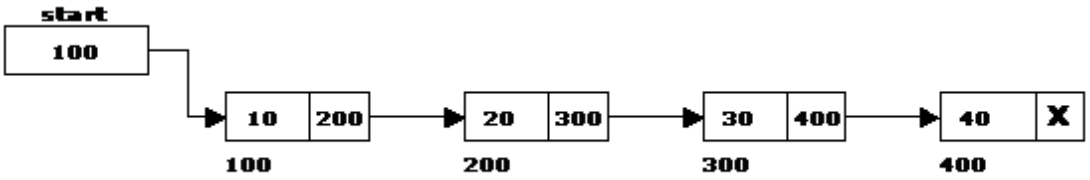
Disadvantages of singly linked list:

1. Backward traversing is not possible in singly linked list.

2. Insertion is easy but deletion take some additional time, because disadvantage of backward traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node



Insertion of a Node:

The new node can then be inserted at three different places namely:

1. Inserting a node at the beginning.
2. Inserting a node at the end.
3. Inserting a node at intermediate position.

Inserting a node at the beginning:

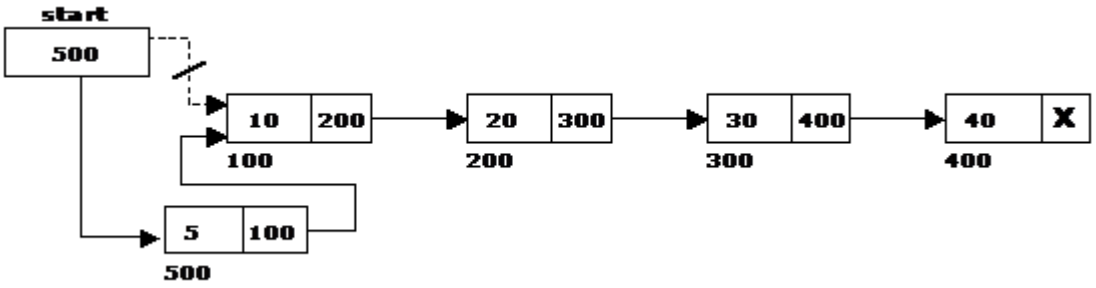
The following steps are to be followed to insert a new node at the beginning of the list:

1. Get the new node using getnode().

newnode = getnode();

2. If the list is empty then start = newnode.
3. If the list is not empty, follow the steps given below:

newnode -> next = start; start = newnode;



Inserting a node at the end:

1. The following steps are followed to insert a new node at the end of the list:
2. Get the new node using getnode()

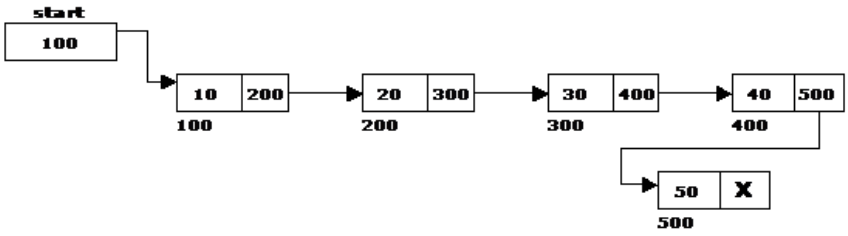
newnode = getnode();

3. If the list is empty then start = newnode.
4. If the list is not empty follow the steps given below:

temp = start;

while(temp -> next != NULL)

1. temp = temp -> next; temp -> next = newnode;



Inserting a node at intermediate position:

1. The following steps are followed, to insert a new node in an intermediate position in the list:

2. Get the new node using getnode().

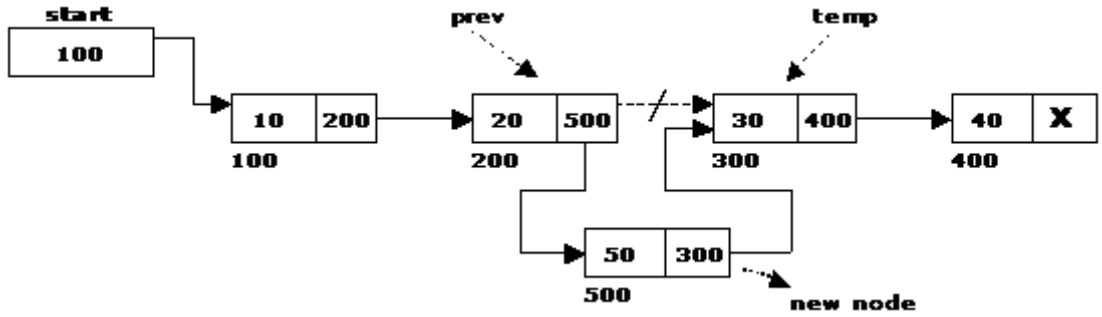
newnode = getnode();

3. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

4. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

5. After reaching the specified position, follow the steps given below:

prev -> next = newnode; newnode -> next = temp;



Deletion of a node:

A node can be deleted from the list from three different places namely.

1. Deleting a node at the beginning.

2. Deleting a node at the end.

3. Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

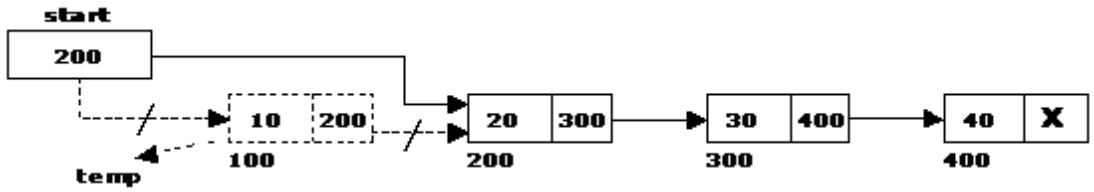
1. If list is empty then display ‘Empty List’ message.

2. If the list is not empty, follow the steps given below:

i. temp = start;

ii. start = start -> next;

iii. free(temp);



Deleting a node at the end:

1. The following steps are followed to delete a node at the end of the list:

2. If list is empty then display 'Empty List' message.

3. If the list is not empty, follow the steps given below:

temp = prev = start; while(temp -> next != NULL)

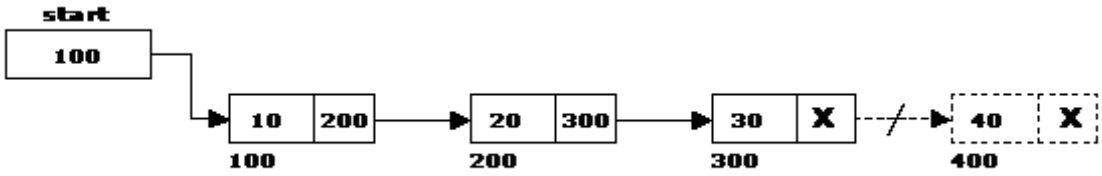
{

1. prev = temp;

2. temp = temp -> next;

}

prev -> next = NULL; free(temp);



Deleting a node at Intermediate position:

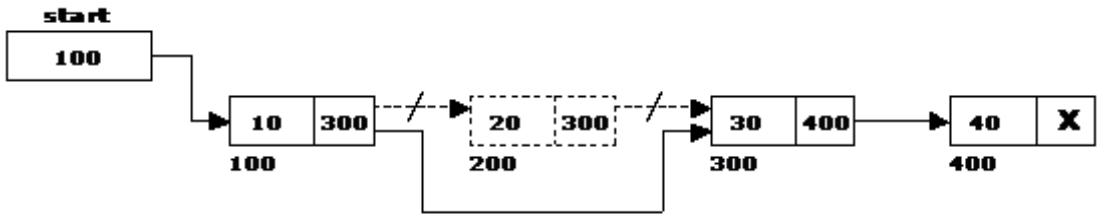
The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

1. If list is empty then display 'Empty List' message

2. If the list is not empty, follow the steps given below.

3. if(pos > 1 && pos < nodectr)

```
{

temp = prev = start; ctr = 1;
while(ctr < pos)
{
temp = temp -> next; ctr++;
}

prev -> next = temp -> next; free(temp);
printf("\n node deleted..");
}
```



Traversal and displaying a list (Left to Right):
Traversing a list involves the following steps:

1. Assign the address of start pointer to a temp pointer.

2. Display the information from the data field of each node.

Source Code:

```
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None

class Sll:
    def __init__(self):
        self.start=None
    def createlist(self):
        n=int(input("enter number of node"))
        for i in range(n):
            data=int(input("enter balue"))
            newnode=Node(data)
            if self.start==None:
                self.start=newnode
            else:
                temp=self.start
                while temp.next!=None:
                    temp=temp.next
                temp.next=newnode
    def insertend(self):
        n=int(input("enter value"))
```

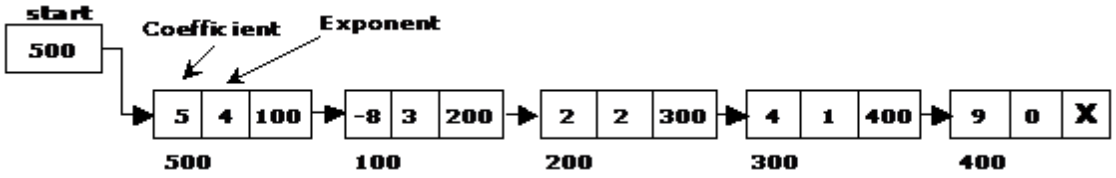
```
        newnode=Node(n)
        if self.start==None:
            self.start=newnode
        else:
            temp=self.start
            while temp.next!=None:
                temp=temp.next
            temp.next=newnode
    def insertmid(self):
        n=int(input("enter value"))
        newnode=Node(n)
        pos=int(input("enter position"))
        c=self.count()
        if self.start==NOne:
            self.start=newnode
        else:
            if pos>1 and pos<=c:
                temp=self.start
                prev=temp
                i=1
                while i<pos:
                    prev=temp
                    temp=temp.next
                    i=i+1
            pre.next=newnode
            newnode.next=temp

    def count(self):
        nc=0
        temp=self.start
        while temp!=None:
            nc+=1
            temp=temp.next
        print("number of nodes=%d" %nc)
        return nc

    def deletemid(self):
        count=1
        if self.start==None:
            print("empty")
        else:
            position=int(input("enter position"))
            c=self.count()
            if position>c:
                print("check position")
            if position>1 and position<c:
                temp=prev=self.start
                while count<position:
```

```
                rev=temp
                temp=temp.next
                count=count+1
            prev.next=temp.next
            del temp
        else:
            print("check position")

def deleteend(self):
    global prev
    if self.start==None:
        print("empty")
    else:
        temp=self.start
        prev=self.start
        while temp.next!=None:
            prev=temp
            temp=temp.next
        prev.next=None
        del temp

def insertbegin(self):
    n=int(input("enter value"))
    newnode=Node(n)
    if self.start==None:
        self.start=newnode
    else:
        temp=self.start
        newnode.next=temp
        self.start=newnode

def deletebegin(self):
    global prev
    if self.start==None:
        print("empty")
    else:
        temp=self.start
        newstart=self.start.next
        del temp
        self.start=newstart

def display(self):
    print("elements in single linked list are:")
    if self.start==None:
        print("empty")
    else:
        temp=self.start
        print("%d" %(temp.data))
```

```
            while temp.next!=None:
                temp=temp.next
                print("%d" %(temp.data))

### OUTSIDE CLASS
def menu():
    print("1. create list \n2. insert begin \n3. insertend \n4. insertmid \n5. deletebegin \n6. deleteend \n7. deletemid \n8. count \n9. display \n10. exit")
def stop():
    print("u r about to terminate program")
    exit(0)
s=Sll()
def default():
    print("check ut input")
menu()
while True:
    menu={
    1: s.createlist,
    2: s.insertbegin,
    3: s.insertend,

    4: s.insertmid,

    5: s.deletebegin,

    6: s.deleteend,

    7: s.deletemid,

    8: s.count,

    9: s.display,

    10: stop}

    option=int(input("enter ur choice"))

    menu.get(option)()

reak; case 10:del_at_mid();

break;

case 11:exit(0);

}

}

}
```

Applications of Single Linked List to Represent Polynomial Expressions:

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials $5x^4 – 8x^3 + 2x^2 + 4x^1 + 9x^0$ illustrates in figure ----



Sparse Matrix Manipulation:

A **sparse matrix** is a matrix populated primarily with zeros as elements of the table.

Example of sparse matrix

[11 22 0 0 0 0 0]

[0 33 44 0 0 0 0]

[0 0 55 66 77 0 0]

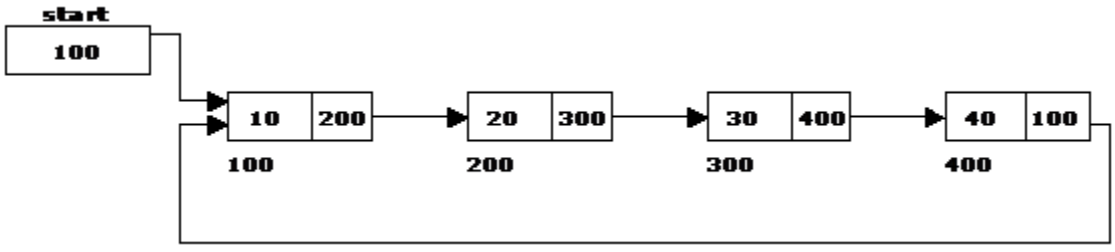
[0 0 0 0 0 88 0]

[0 0 0 0 0 0 99]

The above sparse matrix contains only 9 nonzero elements of the 35, with 26 of those elements as zero. The basic data structure for a matrix is a two-dimensional array. Each entry in the array represents an element $a_{i,j}$ of the matrix and can be accessed by the two indices i and j. Traditionally, i indicates the row number (top-to-bottom), while j indicates the column number (left-to-right) of each element in the table. For an m×n matrix, enough memory to store up to (m×n) entries to represent the matrix is needed.

Circular Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called start pointer always pointing to the first node of the list.



The basic operations in a circular single linked list are:

1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.

Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

1. Get the new node using getnode().

newnode = getnode();

2. If the list is empty, assign new node as start.

start = newnode;

3. If the list is not empty, follow the steps given below:

temp = start;

while(temp -> next != NULL) temp = temp -> next;

temp -> next = newnode;

4. Repeat the above steps 'n' times.

newnode -> next = start;

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

1. Get the new node using getnode().

newnode = getnode();

2. If the list is empty, assign new node as start. start = newnode;

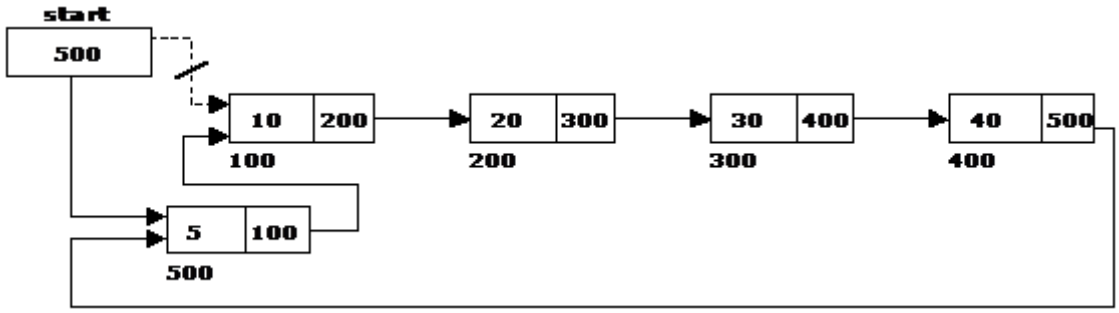
newnode -> next = start;

3. If the list is not empty, follow the steps given below:

last = start;

while(last -> next != start) last = last -> next; newnode -> next = start; start = newnode;

last -> next = start;



Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode().

newnode = getnode();

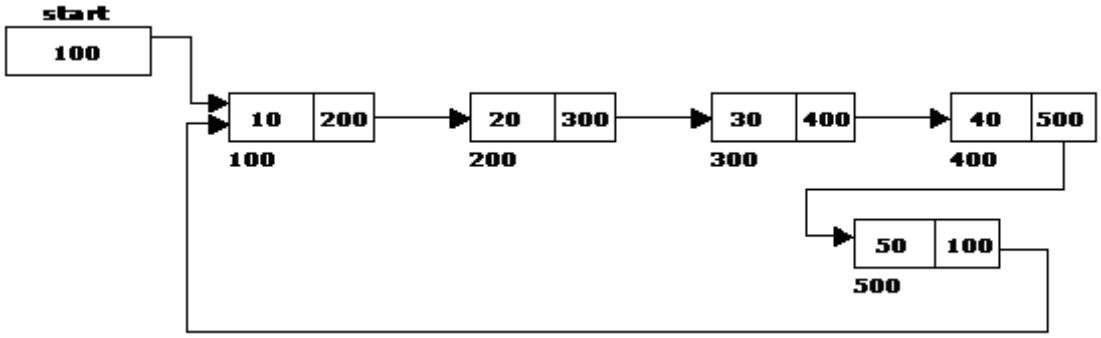
2. If the list is empty, assign new node as start. start = newnode;

newnode -> next = start;

3. If the list is not empty follow the steps given below: temp = start;

while(temp -> next != start) temp = temp -> next;

temp -> next = newnode; newnode -> next = start;



Deleting a node at the beginning:

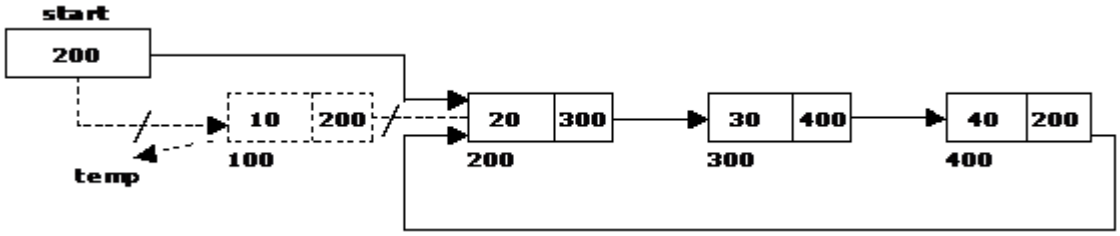
The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message ‘Empty List’.
- If the list is not empty, follow the steps given below: last = temp = start;

while(last -> next != start)

last = last -> next; start = start -> next;

last -> next = start;



Deleting a node at the end:

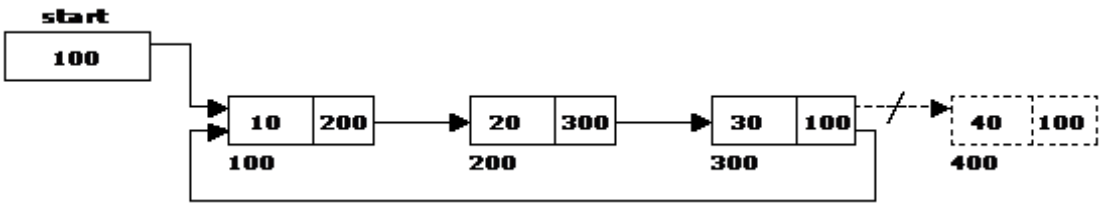
The following steps are followed to delete a node at the end of the list:

1. If the list is empty, display a message 'Empty List'.

2. If the list is not empty, follow the steps given below:

```
temp = start; prev = start;

while(temp -> next != start)

{

prev = temp;

temp = temp -> next;

}

prev -> next = start;
```

3. After deleting the node, if the list is empty then start = NULL.



Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display 'Empty List' message.

2. If the list is not empty, follow the steps given below: temp = start;

```
do

{

temp = temp -> next;

} while(temp != start);
```

Source Code:

```
# Python program to demonstrate circular linked list traversal
 # Structure for a Node
class Node:
        # Constructor to create  a new node
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
        # Constructor to create a empty circular linked list
    def __init__(self):
        self.head = None
     # Function to insert a node at the beginning of a
    # circular linked list
    def push(self, data):
        ptr1 = Node(data)
        temp = self.head
         ptr1.next = self.head

        # If linked list is not None then set the next of
        # last node
        if self.head is not None:
            while(temp.next != self.head):
                temp = temp.next
            temp.next = ptr1

        else:
            ptr1.next = ptr1 # For the first node

        self.head = ptr1

    # Function to print nodes in a given circular linked list
    def printList(self):
        temp = self.head
        if self.head is not None:
            while(True):
                print "%d" %(temp.data),
                temp = temp.next
                if (temp == self.head):
                    break


# Driver program to test above function

# Initialize list as empty
cllist = CircularLinkedList()
```

```
 # Created linked list will be 11->2->56->12
cllist.push(12)
cllist.push(56)
cllist.push(2)
cllist.push(11)

print "Contents of circular Linked List"
cllist.printList()
```

Double Linked List:

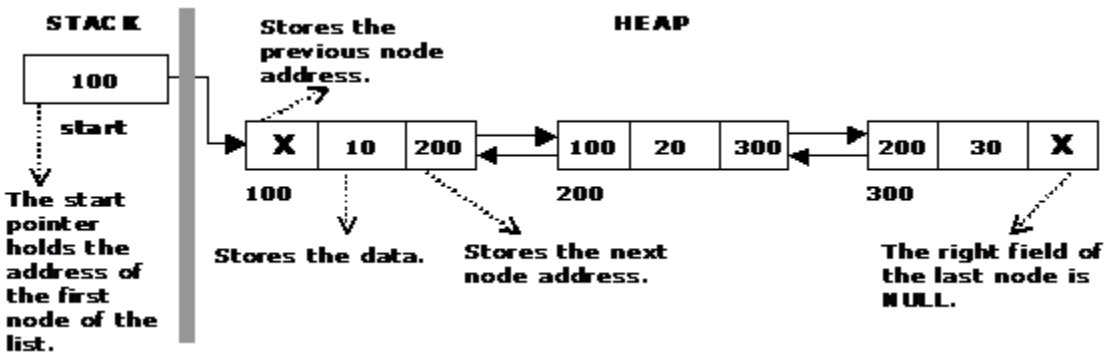
Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node. The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly- linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

1. Left link.
2. Data.
3. Right link.

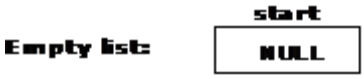
The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data. The basic operations in a double linked list are:

1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.



The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};

typedef struct dlinklist node;
node *start = NULL;
```



Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function.

Creating a Double Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

1. Get the new node using getnode().

newnode =getnode();

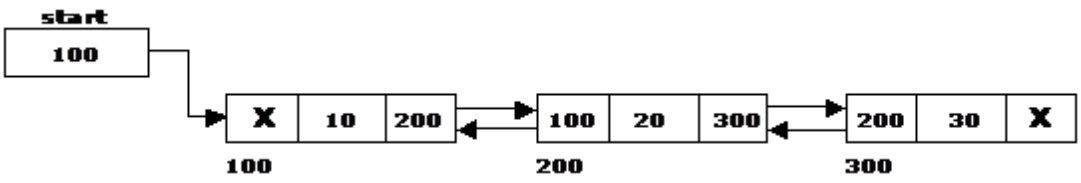
2. If the list is empty then start = newnode.

3. If the list is not empty, follow the steps given below:

i. The left field of the new node is made to point the previous node.

ii. The previous nodes right field must be assigned with address of the new node.

4. Repeat the above steps 'n' times.



Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

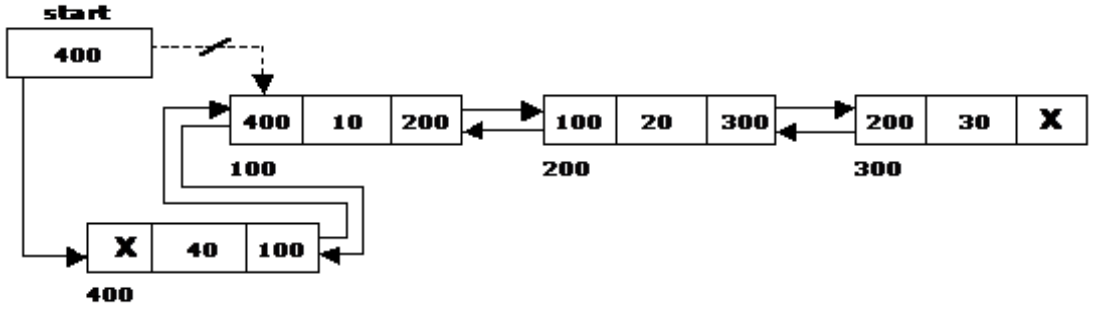
1. Get the new node using getnode().

newnode=getnode();

2. If the list is empty then start = newnode.

3. If the list is not empty, follow the steps given below: newnode -> right = start;

start -> left = newnode; start = newnode;



Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode()

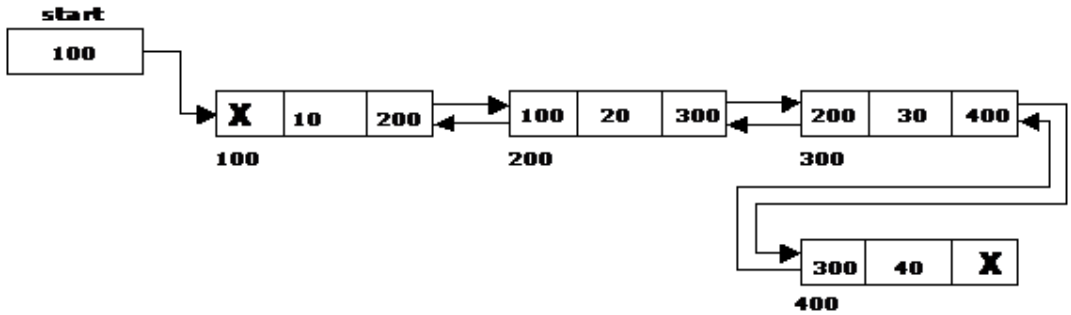
newnode=getnode();

2. If the list is empty then start = newnode.

3. If the list is not empty follow the steps given below: temp = start;

while(temp -> right != NULL) temp = temp -> right;

temp -> right = newnode; newnode -> left = temp;



Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

1. Get the new node using getnode().

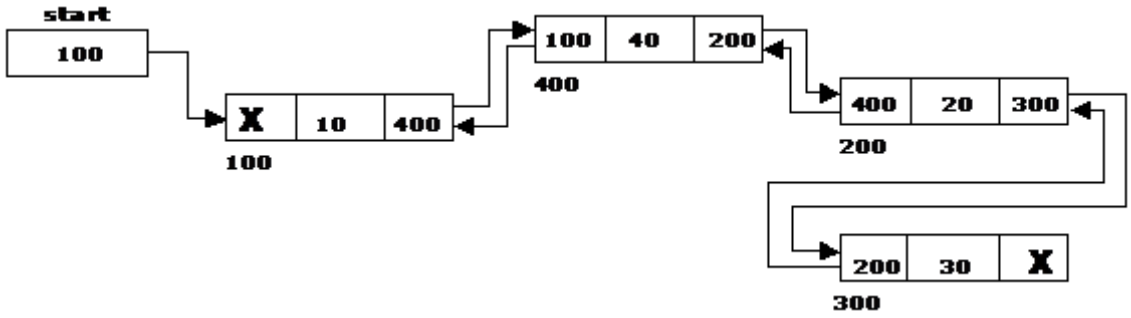
newnode=getnode();

2. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

3. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

4. After reaching the specified position, follow the steps given below: newnode -> left = temp;

newnode -> right = temp -> right; temp -> right -> left = newnode; temp -> right = newnode;



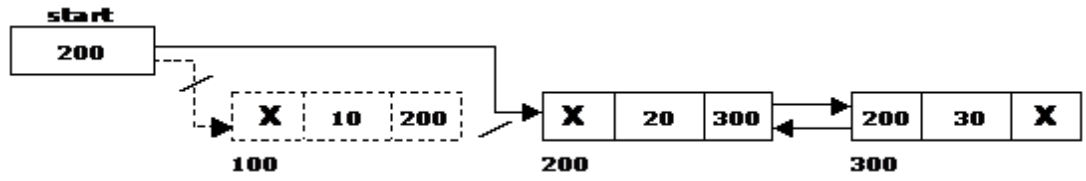
Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display 'Empty List' message.

2. If the list is not empty, follow the steps given below: temp = start;

start = start -> right; start -> left = NULL; free(temp);



Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

1. If list is empty then display 'Empty List' message

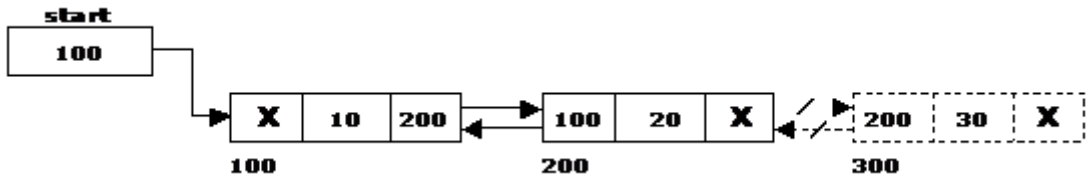
2. If the list is not empty, follow the steps given below: temp = start;

while(temp -> right != NULL)

{

temp = temp -> right;
}

temp -> left -> right = NULL; free(temp);



Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

1. If list is empty then display 'Empty List' message.

2. If the list is not empty, follow the steps given below:

i. Get the position of the node to delete.

ii. Ensure that the specified position is in between first node and last node. If not, specified position is invalid.

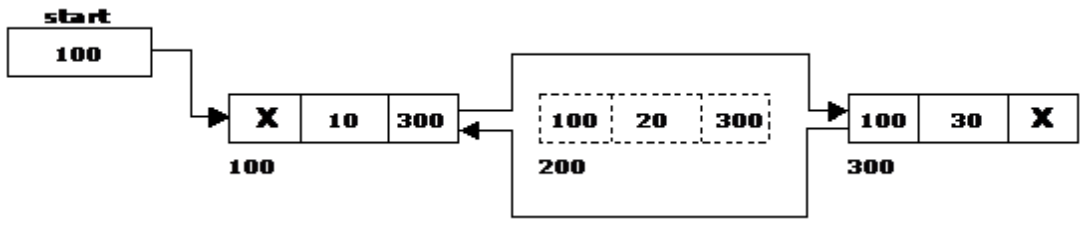
iii. Then perform the following steps:

if(pos > 1 && pos < nodectr)
{

temp = start; i = 1;
while(i < pos)
{

temp = temp -> right; i++;
}

temp -> right -> left = temp -> left; temp -> left -> right = temp -> right; free(temp);
printf("\n node deleted..");
}



Traversal and displaying a list (Left to Right):

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display ‘Empty List’ message.

2. If the list is not empty, follow the steps given below:

temp = start; while(temp != NULL)

{

print temp -> data; temp = temp -> right;

}

Traversal and displaying a list (Right to Left):

The following steps are followed, to traverse a list from right to left:

1. If list is empty then display ‘Empty List’ message.

2. If the list is not empty, follow the steps given below: temp = start;

while(temp -> right != NULL) temp = temp -> right;
while(temp != NULL)
{

print temp -> data; temp = temp -> left;
}

Source Code:

```
class
Node():
        def __init__(self, next_node=None, previous_node=None, data=None):
            self.next_node = next_node
            self.previous_node = previous_node
            self.data = data
    class LinkedList():
        def __init__(self, node):
            assert isinstance(node, Node)
            self.first_node = node
            self.last_node = node
        def push(self, node):
            '''Pushes the node <node> at the "front" of the ll
            '''
            node.next_node = self.first_node
            node.previous_node = None
            self.first_node.previous_node = node
            self.first_node = node
        def pop(self):
```

```
        '''Pops the last node out of the list'''
        old_last_node = self.last_node
        to_be_last = self.last_node.previous_node
        to_be_last.next_node = None
        old_last_node.previous_node = None
        # Set the last node to the "to_be_last"
        self.previous_node = to_be_last
        return old_last_node
    def remove(self, node):
        '''Removes and returns node, and connects the previous and next
        nicely
        '''
        next_node = node.next_node
        previous_node = node.previous_node
        previous_node.next_node = next_node
        next_node.previous_node = previous_node
        # Make it "free"
        node.next_node = node.previous_node = None
        return node
    def __str__(self):
        next_node = self.first_node
        s = ""
        while next_node:
            s += "--({:0>2d})--\n".format(next_node.data)
            next_node = next_node.next_node
        return s
        return
node1 = Node(data=1)
linked_list = LinkedList(node1)
for i in xrange(10):
    if i == 5:
        node5 = Node(data=5)
        linked_list.push(node5)
    else:
        linked_list.push(Node(data=i))
print linked_list
print "popping"
print linked_list.pop().data
print "\n\n"
print linked_list
print "\n\n"
linked_list.push(Node(data=10))
print "\n\n"
print linked_list
linked_list.remove(node5)
print "\n\n"
print linked_list
```

Stack using linked list:

```
# Python program for linked list implementation of stack
 # Class to represent a node
class StackNode:
     # Constructor to initialize a node
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
         # Constructor to initialize the root of linked list
    def __init__(self):
        self.root = None
     def isEmpty(self):
        return True if self.root is None else  False
     def push(self, data):
        newNode = StackNode(data)
        newNode.next = self.root
        self.root = newNode
        print ("%d pushed to stack" %(data))

    def pop(self):
        if (self.isEmpty()):
            return float("-inf")
        temp = self.root
        self.root = self.root.next
        popped = temp.data
        return popped

    def peek(self):
        if self.isEmpty():
            return float("-inf")
        return self.root.data

# Driver program to test above class
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)

print ("%d popped from stack" %(stack.pop()))
print ("Top element is %d " %(stack.peek()))
```

Queue using linked list.

```
# program to implement queues using linked list

class stack(object):
    def __init__(self, value=None, next=None):
        self.value = value
        self.next = next

    def push(self, value):
        oldstack = stack(self.value, self.next)
        self.value = value
        self.next = oldstack

    def pop(self):
        outputval = self.value
        self.value, self.next = self.next.value, self.next.next
        return outputval
    def size(self):
        numelements = 0
        nextval = self.next
        while (nextval != None):
            nextval = nextval.next
            numelements += 1
        return numelements

    def isEmpty(self):
        if (self.value == None):
            return True
        else:
            return False
class queue(object):
    def __init__(self):
        self.stack1 = stack()
        self.stack2 = stack()
    def fillStack2(self):
        if (self.stack2.isEmpty()):
            while (not self.stack1.isEmpty()):
                self.stack2.push(self.stack1.pop())
    def push(self, value):
        self.stack1.push(value)
        self.fillStack2()
        return value
    def pop(self):
        if (self.stack2.isEmpty()):
            raise NameError('Queue.Empty')
        outputval = self.stack2.pop()
        self.fillStack2()
        return outputval
```

```
a = queue()
print ("pushed element is",a.push(2))
print ("pushed element is",a.push(3))
print ("pushed element is",a.push(4))
print ("pushed element is",a.push(12))
print ("----------------------------")
print ("popped element is",a.pop())
print ("popped element is",a.pop())
print ("popped element is",a.pop())
print ("----------------------------")
print ("pushed element is",a.push(5))
print ("----------------------------")
print ("popped element is",a.pop())
print ("popped element is",a.pop())
```

Module – IV
Non Linear Data Structures

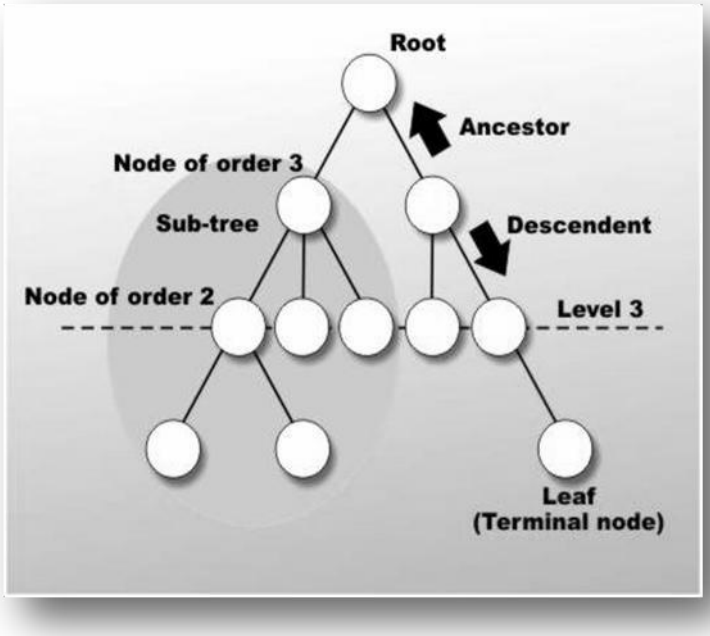
Basic Concept:

A tree is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a sub-tree of the root.

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

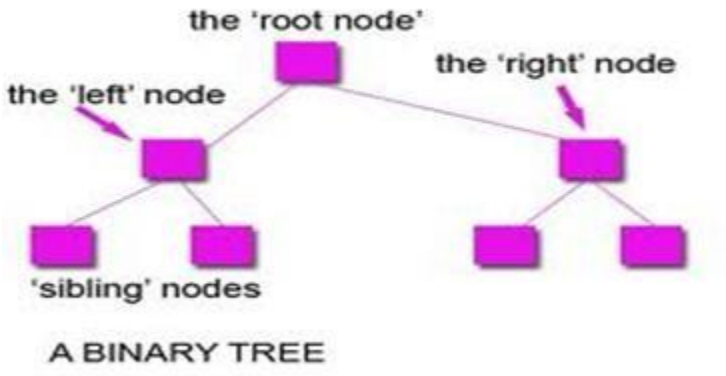
- If T is not empty, T has a special tree called the root that has no parent.
- Each node v of T different than the root has a unique parent node w; each node with parent w is a child of w.

Tree nodes have many useful properties. The depth of a node is the length of the path (or the number of edges) from the root to that node. The height of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A leaf node has no children -- its only path is up to its parent.



Binary Tree:

In a binary tree, each node can have at most two children. A binary tree is either empty or consists of a node called the root together with two binary trees called the left sub-tree and the right sub-tree.



Tree Terminology:

Leaf node

A node with no children is called a leaf (or external node). A node which is not a leaf is called an internal node.

Path: A sequence of nodes n_1, n_2, . . ., n_k, such that n_i is the parent of n_{i+1} for i = 1, 2,. . ., k - 1. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

Siblings: The children of the same parent are called siblings.

Ancestor and Descendent If there is a path from node A to node B, then A is called an ancestor of B and B is called a descendent of A.

Sub-tree: Any node of a tree, with all of its descendants is a sub-tree.

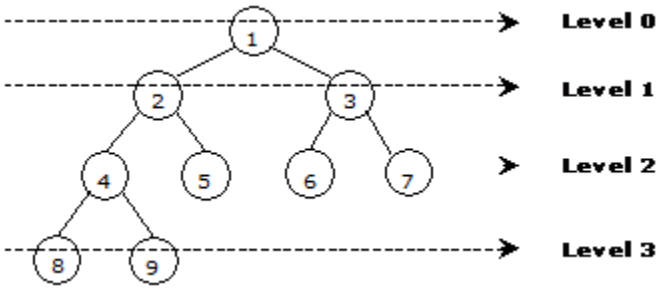
Level: The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent.

The maximum number of nodes at any level is 2^n.

Height:The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree.

Depth:The depth of a node is the number of nodes along the path from the root to that node.

Assigning level numbers and Numbering of nodes for a binary tree: The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent.



Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then

a. Maximum number of leaves = 2^h

b. Maximum number of nodes = $2^{h+1} - 1$

2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.

3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.

4. The total number of edges in a full binary tree with n node is n - 1.

Binary Tree Representation:

1. Array Representation of Binary Tree

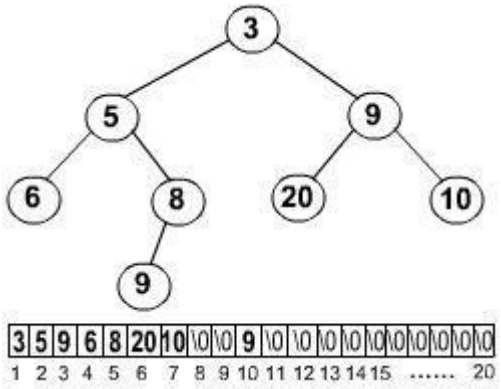
2. Linked List Representation of Binary Tree

Array Representation of Binary Tree:

A single array can be used to represent a binary tree.

For these nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index i is put into the array as its i^{th} element.

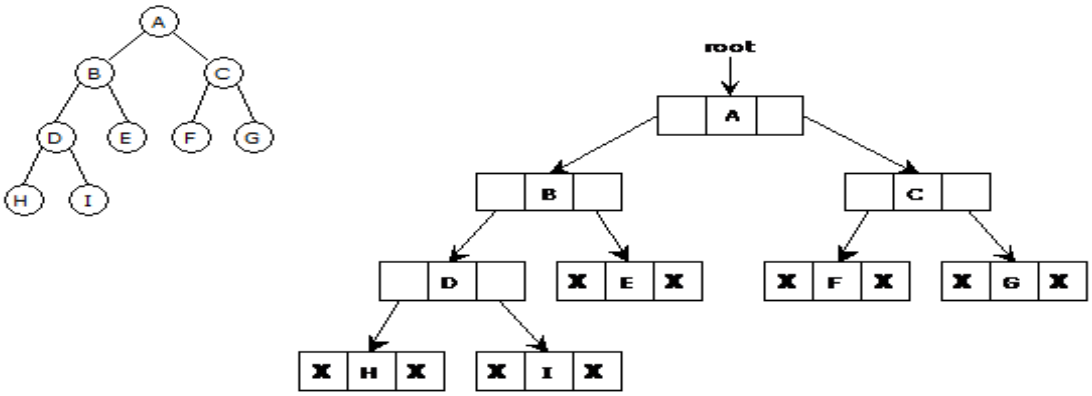
In the figure shown below the nodes of binary tree are numbered according to the given scheme.



The figure shows how a binary tree is represented as an array. The root 3 is the 0^{th} element while its leftchild 5 is the 1^{st} element of the array. Node 6 does not have any child so its children i.e. 7^{th} and 8^{th} element of the array are shown as a Null value.

It is found that if n is the number or index of a node, then its left child occurs at $(2n + 1)^{th}$ position and right child at (2n + 2)th position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

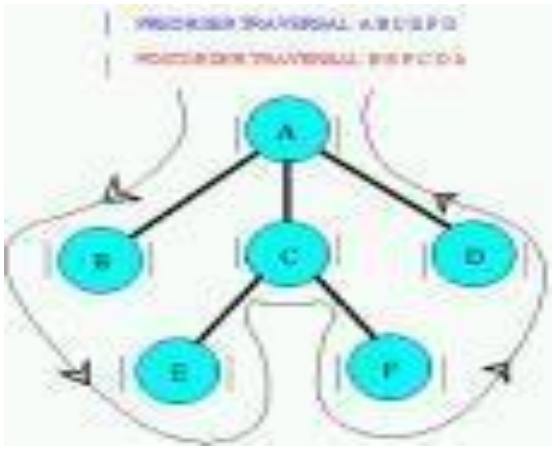
Linked Representation of Binary Tree (Pointer based):



Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

Binary Tree Traversal:

Traversal of a binary tree means to visit each node in the tree exactly once.



In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. Th ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

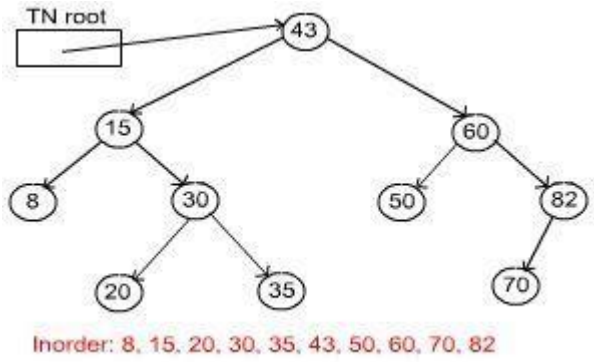
In all of them we do not require to do anything to traverse an empty tree. All the traversal methods are base functions since a binary tree is itself recursive as every child of a node in a binary tree is itself a binary tree.

In-order Traversal:

To traverse a non empty tree in in-order the following steps are followed recursively.

- Visit the Root
- Traverse the left sub-tree
- Traverse the right sub-tree

The in-order traversal of the tree shown below is as follows.



Algorithms:-

In-order Traversal:

Algorithm In-order(tree)

1. Traverse the left sub-tree, i.e., call In-order(left-sub-tree)
2. Visit the root.
3. Traverse the right sub-tree, i.e., call In-order(right-sub-tree)

So the function calls itself recursively and carries on the traversal.

Pre-order Traversal:

Algorithm Pre-order(tree)

1. Visit the root.
2. Traverse the left sub-tree, i.e., call Pre-order(left-sub-tree)
3. Traverse the right sub-tree, i.e., call Pre-order(right-sub-tree)

Post-order Traversal:

Algorithm Post-order(tree)

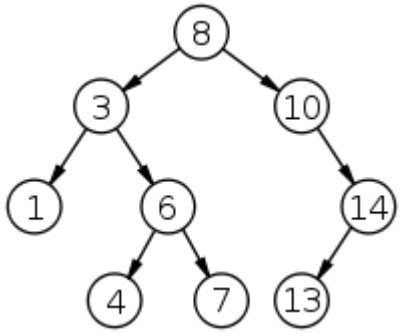
1. Traverse the left sub-tree, i.e., call Post-order(left-sub-tree)
2. Traverse the right sub-tree, i.e., call Post-order(right-sub-tree)
3. Visit the root.

Time Complexity: O(n^2).

Binary Search Tree:

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left sub-tree of a node contains only nodes with keys less than the node's key.
- The right sub-tree of a node contains only nodes with keys greater than the node's key.
- The left and right sub-tree each must also be a binary search tree. There must be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Searching a key

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right sub-tree of root node. Otherwise we recur for left sub-tree.

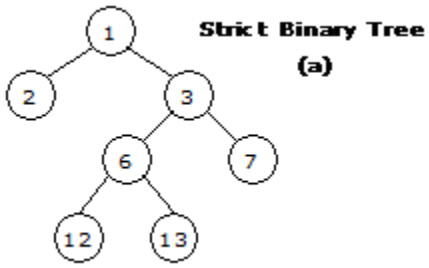
```
# A utility function to search a given key in BST

def search(root,key)

    # Base Cases: root is null or key is present at root

    if root is None or root.val == key:

        return root
```

```
    # Key is greater than root's key
    if root.val < key:
        return search(root.right,key)
      # Key is smaller than root's key
    return search(root.left,key)
```

Tree Variants

Strictly Binary tree:

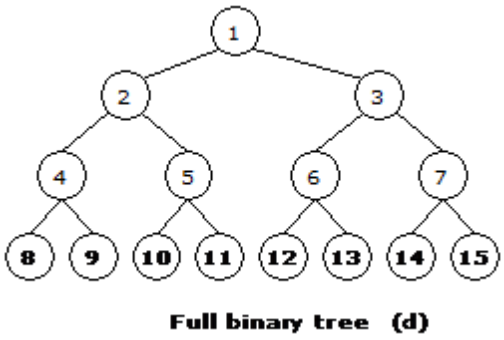
If every non-leaf node in a binary tree has nonempty left and right sub-trees, the tree is termed a strictly binary tree. Thus the tree of figure 7.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1 nodes.



Full Binary Tree:

A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height h has 2^{h+1} - 1 nodes. A full binary tree of height h is a strictly binary tree all of whose leaves are at level h.

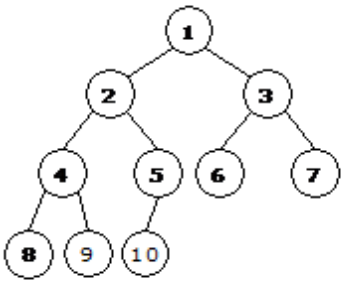


For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

Complete Binary Tree:

A binary tree with n nodes is said to be complete if it contains all the first n nodes of the above numbering scheme.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.



Complete binary tree (c)

Perfect Binary Tree:

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

Following are examples of Perfect Binary Trees.

```
          18
       /     \
     15       30
    / \      / \
  40   50   100  40
```

```
          18
       /     \
     15       30
```

A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has $2^h - 1$ node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree:

A binary tree is balanced if height of the tree is O(Log n) where n is number of nodes. For Example, AVL tree maintain O(Log n) height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.

Application of Trees:

1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system

/ <-- root

/ \

... home

/ \

ugrad course

/ / | \

... cs101 cs112 cs113

2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log_n)$ for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(logn) for insertion/deletion.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don‘t have an upper limit on number of nodes as nodes are linked using pointers.

The following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

Graphs:

Basic Concepts:

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.

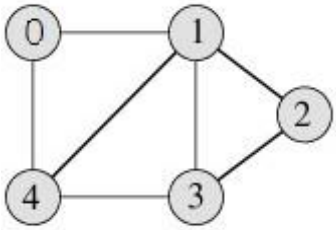
2. A finite set of ordered pair of the form (u, v) called as edge.

The pair is ordered because (u, v) is not same as (v, u) in case of directed graph (di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graph and its representations:

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

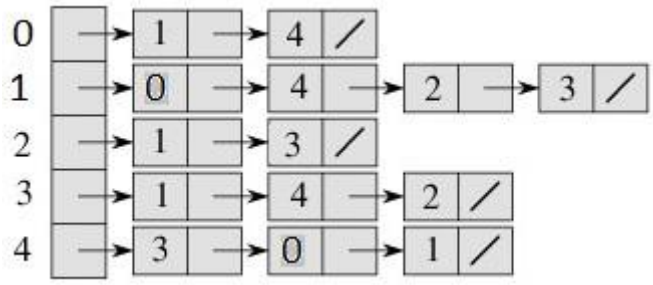
Adjacency Matrix Representation of the above graph

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex _u‘ to vertex _v‘ are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

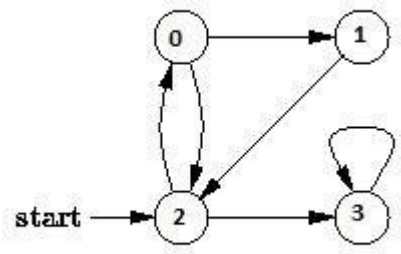


Adjacency List Representation of the above Graph

Breadth First Traversal for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don‘t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Breadth First Traversal of the following graph is 2, 0, 3, 1.



Algorithm: Breadth-First Search Traversal BFS(V, E, s)

for each u in V − {s}

do color[u] ← WHITE d[u] ← infinity π[u] ← NIL color[s] ← GRAY

d[s] ← 0 π[s] ← NIL Q ← {}

ENQUEUE(Q, s)

while Q is non-empty

do u ← DEQUEUE(Q)

for each v adjacent to u

do if color[v] ← WHITE

then color[v] ← GRAY d[v] ← d[u] + 1

π[v] ← u

ENQUEUE(Q, v) DEQUEUE(Q)

color[u] ← BLACK

Source Code:

```
# Program to print BFS traversal from a given source
# vertex. BFS(int s) traverses vertices reachable
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
```

```
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            print (s)

            # Get all adjacent vertices of the dequeued
            # vertex s. If a adjacent has not been visited,
            # then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print  ("Following is Breadth First Traversal (starting from vertex 2)")
g.BFS(2)
```

Applications of Breadth First Traversal

1) Shortest Path and Minimum Spanning Tree for unweighted graph In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) Crawlers in Search Engines: Crawlers build index using Bread First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance ‗k' from a person using Breadth First Search till ‗k' levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney's algorithm.

8) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) Ford–Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

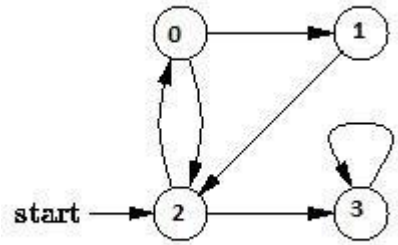
10) To test if a graph is Bipartite We can either use Breadth First or Depth First Traversal.

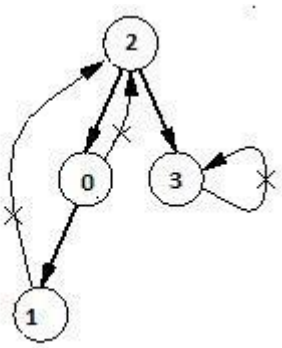
11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Depth First Traversal for a Graph

Depth first search for a graph is similar to depth first traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Depth First Traversal of the following graph is 2, 0, 1, 3





Algorithm Depth-First Search

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u, v) is explored. The following pseudocode for DFS uses a global timestamp time.

DFS (V, E)

for each vertex u in V[G]

do color[u] ← WHITE π[u] ← NIL

time ← 0

for each vertex u in V[G]

do if color[u] ← WHITE

then DFS-Visit(u)

DFS-Visit(u)

color[u] ← GRAY time ← time + 1 d[u] ← time

for each vertex v adjacent to u

do if color[v] ← WHITE

then π[v] ← u

DFS-Visit(v) color[u] ← BLACK time ← time + 1

f[u] ← time

Source Code:

```
# Python program to print DFS traversal from a
# given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:
```

```
    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self,v,visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print (v),

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)


    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self,v):

        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))

        # Call the recursive helper function to print
        # DFS traversal
        self.DFSUtil(v,visited)


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph. Following are the problems that use DFS as a building block.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

4) Topological Sorting

5) To test if a graph is bipartite

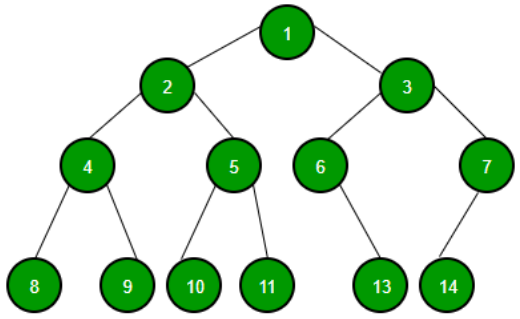
6) Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

Module – V

Binary Trees and Hashing

Binary Trees

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

Properties of binary tree

1) The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}.
Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.
This can be proved by induction.
For root, l = 1, number of nodes = $2^{1-1} = 1$
Assume that maximum number of nodes on level l is 2^{l-1}
Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$

2) Maximum number of nodes in a binary tree of height 'h' is $2^h – 1$.
Here height of a tree is maximum number of nodes on root to leaf path. Height of a tree with single node is considered as 1.
This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^{h-1}$. This is a simple geometric series with h terms and sum of this series is $2^h – 1$.
In some books, height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} – 1$

3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is ? $Log_2(N+1)$?

This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes ? $Log_2(N+1)$? – 1

4) A Binary Tree with L leaves has at least ? Log_2L ? + 1 levels

A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

$L <= 2^{l-1}$ [From Point 1]

l = ? Log_2L ? + 1

where l is the minimum number of levels.

5) In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

L = T + 1

Where L = Number of leaf nodes

T = Number of internal nodes with two children

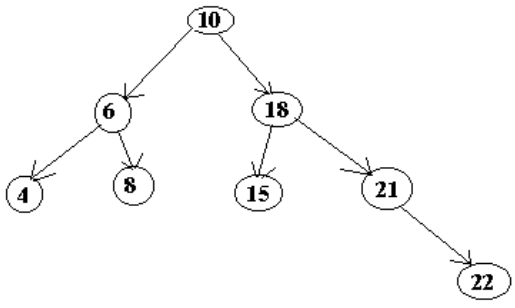
Operations on binary trees

A binary tree is a binary search tree (BST) if and only if an inorder traversal of the binary tree results in a sorted sequence. The idea of a binary search tree is that data is stored according to an order, so that it can be retrieved very efficiently.

A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are < (less) than the key in its parent node
3. The keys in the right subtree > (greater) than the key in its parent node
4. Duplicate node keys are not allowed.

Here is an example of a BST



Exercise : Draw the binary tree which would be created by inserting the following numbers in the order given

50 30 25 75 82 28 63 70 4 43 74 35

If the BST is built in a "balanced" fashion, then BST provides log time access to each element. Consider an arbitrary BST of the height k. The total possible number of nodes is given by

$2^{k+1} - 1$

In order to find a particular node we need to perform one comparison on each level, or maximum of(k+1) total. Now, assume that we know the number of nodes and we want to figure out the number of comparisons. We have to solve the following equation with respect to k:

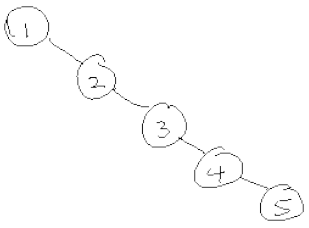
Assume that we have a "balanced" tree with n nodes. If the maximum number of comparisons to find an entry is (k+1), where k is the height, we have

$2^{K+1} - 1 = n$

we obtain

$k = \log_2(n+1) - 1 = O(\log_2 n)$

This means, that a "balanced" BST with n nodes has a maximum order of log(n) levels, and thus it takes at most log(n) comparisons to find a particular node. This is the most important fact you need to know about BSTs. But building a BST as a balanced tree is not a trivial task. If the data is randomly distributed, then we can expect that a tree can be "almost" balanced, or there is a good probability that it would be. However, if the data already has a pattern, then just naïve insertion into a BST will result in unbalanced trees. For example, if we just insert the data 1, 2, 3, 4, 5 into a BST in the order they come, we will end up with a tree that looks like this:



Binary search trees work well for many applications (one of them is a dictionary or help browser). But they can be limiting because of their bad worst-case performance height = O(# nodes). Imagine a binary search tree created from a list that is already sorted.

Clearly, the tree will grow to the right or to the left. A binary search tree with this worst-case structure is no more efficient than a regular linked list. A great care needs to be taken in order to keep the tree as balanced as possible. There are many techniques for balancing a tree including AVL trees, and Splay Trees.

BST OPERATIONS

There are a number of operations on BST's that are important to understand.

Inserting a node

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A definition is as follows.

Insert(N, T) = N if T is empty

= insert(N, T.left) if $N < T$

= insert(N, T.right) if $N > T$

Searching for a node

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on $N < T$ or $N > T$. A recursive definition is as follows.

Search should return a true or false, depending on the node is found or not.

Search(N, T) = false if T is empty

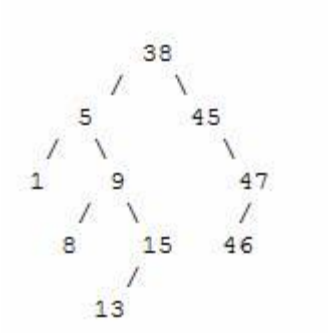
= true if T = N

= search(N, T.left) if $N < T$

= search(N, T.right) if $N > T$

Deleting a node

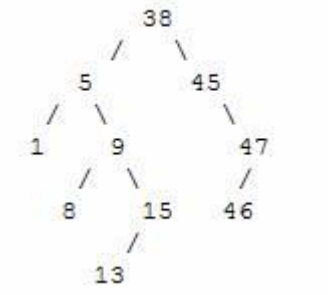
A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree



could result in losing sub trees that are rooted at 1 and 9. Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with just one child? What if the node is an internal node (with two children). The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

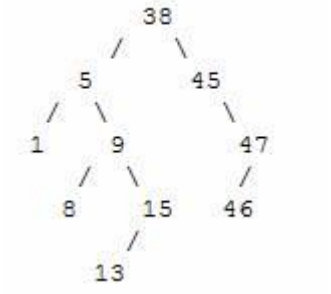
Case 1 : The node to delete is a leaf node

This is a very easy case. Just delete the node. We are done



Case 2 : The node to delete is a node with one child.

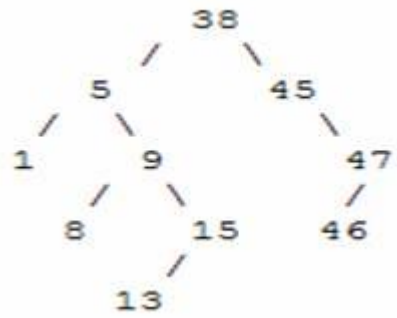
This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.



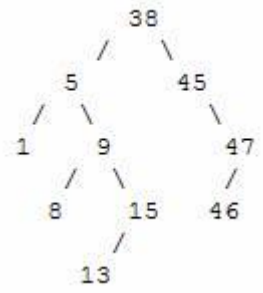
Case 3: The node to delete is a node with two children

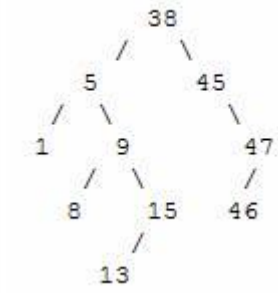
This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)

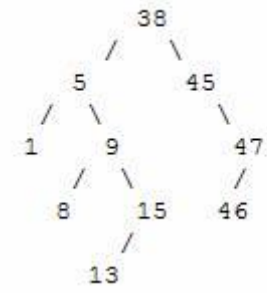
Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, the find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property. For example, consider the following tree and suppose we need to delete the root 38.



Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree (45) and replace the root with that node and then delete that node. The following set of images demonstrates this process.



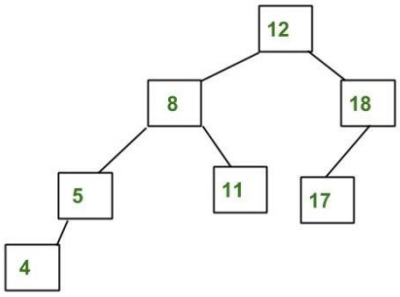




Balanced Search Trees

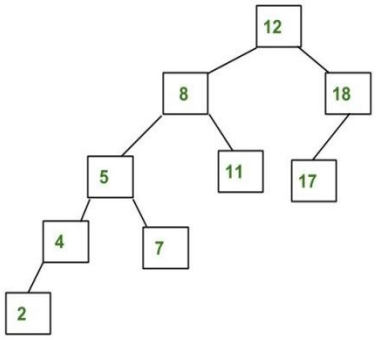
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

Insertion

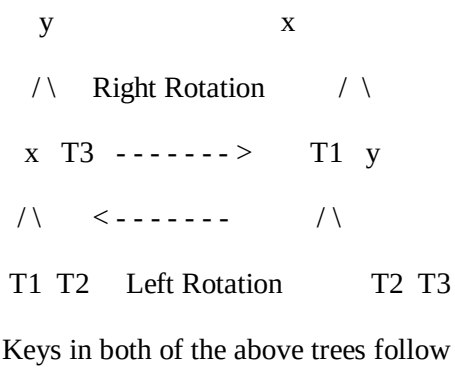
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1) Left Rotation
2) Right Rotation

T1, T2 and T3 are subtrees of the tree

rooted with y (on the left side) or x (on

the right side)



Keys in both of the above trees follow the

following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

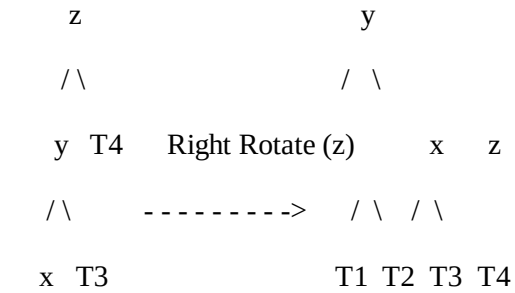
So BST property is not violated anywhere.

Steps to follow for insertion
Let the newly inserted node be w
1) Perform standard BST insert for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

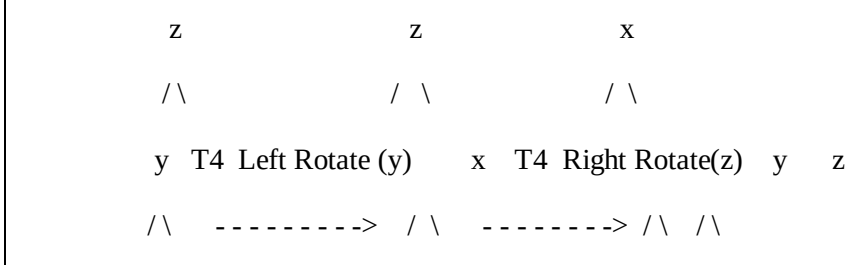
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See this video lecture for proof)

a) Left Left Case

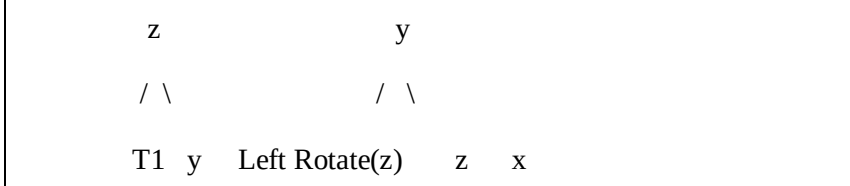
T1, T2, T3 and T4 are subtrees.



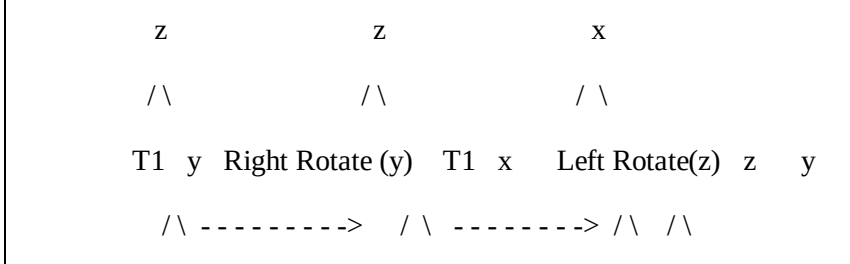
b) Left Right Case



c) Right Right Case

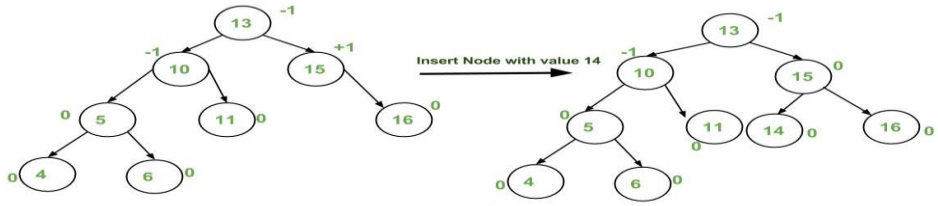


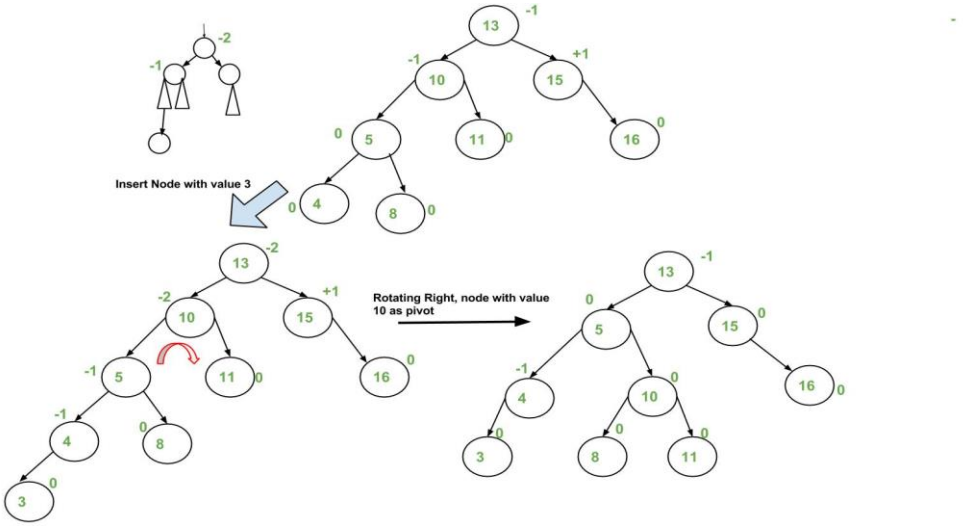
d) Right Left Case

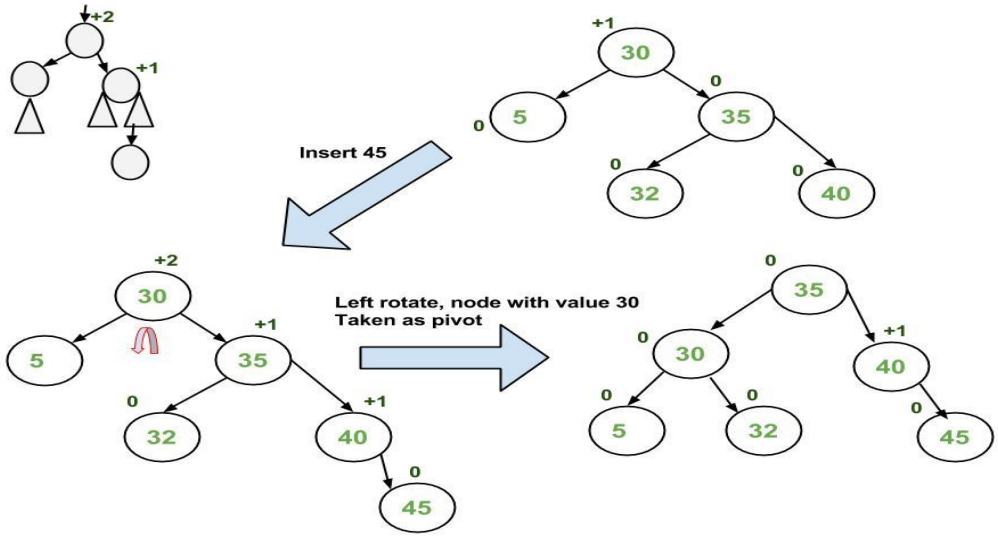


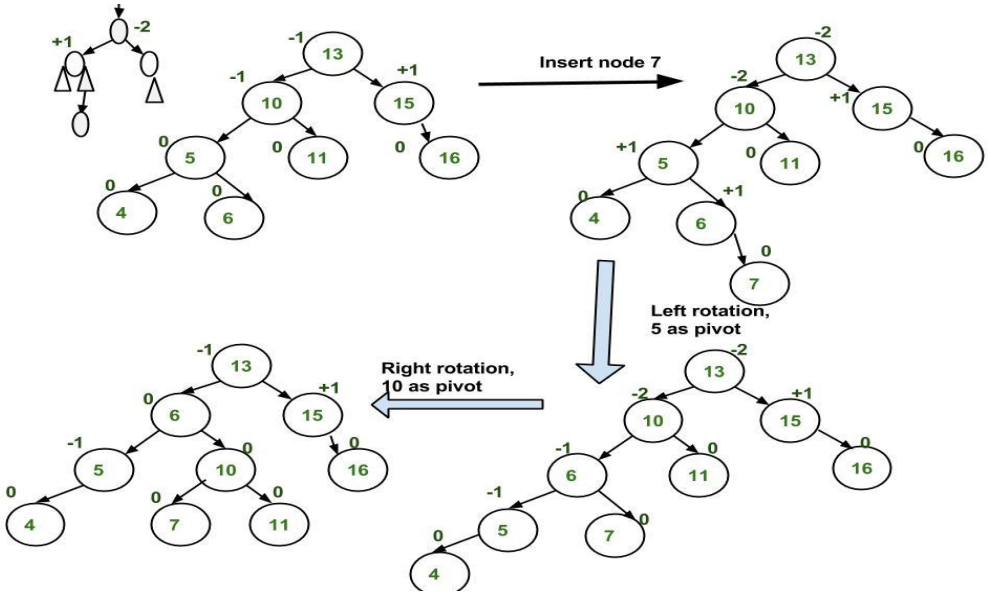
Insertion

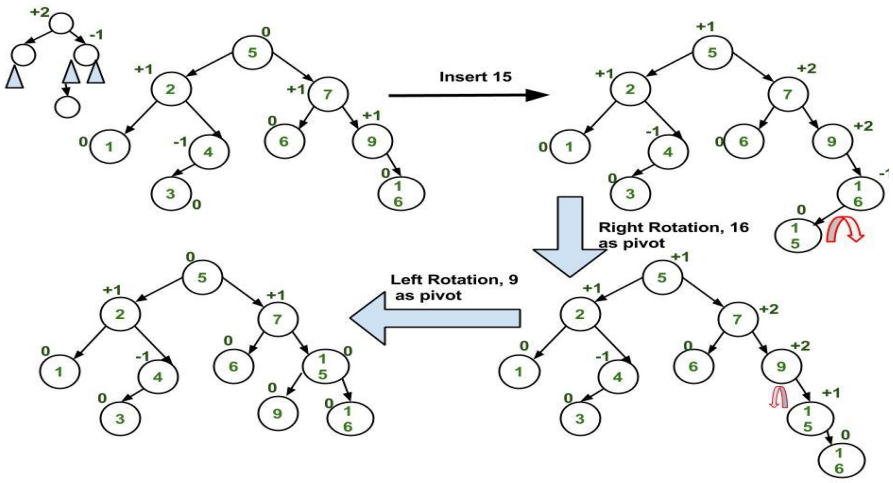
Examples:


Insert Node with value 14


Insert Node with value 3
Rotating Right, node with value 10 as pivot


Insert 45
Left rotate, node with value 30 Taken as pivot





Implementation

Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

1) Perform the normal BST insertion.

2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.

3) Get the balance factor (left subtree height – right subtree height) of the current node.

4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.

5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right

Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

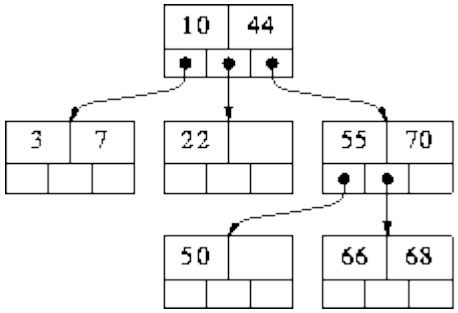
M way trees:

A binary search tree has one value in each node and two subtrees. This notion easily generalizes to an M-way search tree, which has (M-1) values per node and M subtrees. M is called the degreeof the tree. A binary search tree, therefore, has degree 2.

In fact, it is not necessary for every node to contain exactly (M-1) values and have exactly M subtrees. In an M-way subtree a node can have anywhere from 1 to (M-1) values, and the number of (non-empty) subtrees can range from 0 (for a leaf) to 1+(the number of values). M is thus a fixed upper limit on how much data can be stored in a node.

The values in a node are stored in ascending order, V1 < V2 < ... Vk (k <= M-1) and the subtrees are placed between adjacent values, with one additional subtree at each end. We can thus associate with each value a `left' and `right' subtree, with the right subtree of Vi being the same as the left subtree of V(i+1). All the values in V1's left subtree are less than V1 ; all the values in Vk's subtree are greater than Vk; and all the values in the subtree between V(i) and V(i+1) are greater than V(i) and less than V(i+1).

For example, here is a 3-way search tree:



In our examples it will be convenient to illustrate M-way trees using a small value of M. But bear in mind that, in practice, M is usually very large. Each node corresponds to a physical block on disk, and M represents the maximum number of data items that can be stored in a single block. M is maximized in order to speedup processing: to move from one node to another involves reading a block from disk - a very slow operation compared to moving around a data structure stored in memory.

The algorithm for searching for a value in an M-way search tree is the obvious generalization of the algorithm for searching in a binary search tree. If we are searching for value X are and currently at node consisting of values V1...Vk, there are four possible cases that can arise:

1. If X < V1, recursively search for X in V1's left subtree.

2. If X > Vk, recursively search for X in Vk's right subtree.

3. If X=Vi, for some i, then we are done (X has been found).

4. the only remaining possibility is that, for some i, Vi < X < V(i+1). In this case recursively search for X in the subtree that is in between Vi and V(i+1).

For example, suppose we were searching for 68 in the tree above. At the root, case (2) would apply, so we would continue the search in V2's right subtree. At the root of this subtree, case (4) applies, 68 is between V1=55 and V2=70, so we would continue to search in the subtree between them. Now case (3) applies, 68=V2, so we are done. If we had been searching for 69, exactly the same processing would have occurred down to the last node. At that point, case (2) would apply, but the subtree we want to search in is empty. Therefore we conclude that 69 is not in the tree.

Other the algorithms for binary search trees - insertion and deletion - generalize in a similar way. As with binary search trees, inserting values in ascending order will result in a degenerate M-way search tree; i.e. a tree whose height is O(N) instead of O(logN). This is a problem because all the important operations are O(height), and it is our aim to make them O(logN). One solution to this problem is to force the tree to be height-balanced.

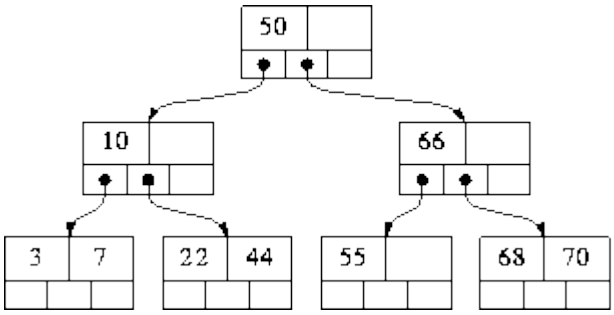
B Trees:

A B-tree is an M-way search tree with two special properties:

1. It is perfectly balanced: every leaf node is at the same depth.

2. Every node, except perhaps the root, is at least half-full, i.e. contains M/2 or more values (of course, it cannot contain more than M-1 values). The root may have any number of values (1 to M-1).

Here is a 3-way B-tree containing the same values:



Hashing and Collision:

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.

2. Search a phone number and fetch the information.

3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.

2. Linked List of phone numbers and records.

3. Balanced binary search tree with phone numbers as keys.

4. Direct Access Table.

For arrays and linked lists, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With balanced binary search tree, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a direct access table where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.
This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. Hashing is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
A good hash function should have following properties
1) Efficiently computable.
2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- Chaining: The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

- Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Applications of hashing:

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc.

There are many other applications of hashing, including modern day cryptography hash functions. Some of these applications are listed below:

- Message Digest
- Password Verification
- Data Structures(Programming Languages)
- Compiler Operation
- Rabin-Karp Algortithm
- Linking File name and path together

Message Digest:

This is an application of cryptographic Hash Functions. Cryptographic hash functions are the functions which produce an output from which reaching the input is close to impossible. This property of hash functions is called irreversibility.

Lets take an Example:

Suppose you have to store your files on any of the cloud services available. You have to be sure that the files that you store are not tampered by any third party. You do it by computing “hash” of that file using a Cryptographic hash algorithm. One of the common cryptographic hash algorithms is SHA 256. The hash thus computed has a maximum size of 32 bytes. So a computing the hash of large number of files will not be a problem. You save these hashes on your local machine.

Now, when you download the files, you compute the hash again. Then you match it with the previous hash computed. Therefore, you know whether your files were tampered or not. If anybody tamper with the file, the hash value of the file will definitely change. Tampering the file without changing the hash is nearly impossible.

Password Verification

Cryptographic hash functions are very commonly used in password verification. Let’s understand this using an Example:

When you use any online website which requires a user login, you enter your E-mail and password to authenticate that the account you are trying to use belongs to you. When the password is entered, a hash of the password is computed which is then sent to the server for verification of the password.

The passwords stored on the server are actually computed hash values of the original passwords. This is done to ensure that when the password is sent from client to server, no sniffing is there.

Data Structures(Programming Languages):

Various programming languages have hash table based Data Structures. The basic idea is to create a key-value pair where key is supposed to be a unique value, whereas value can be same for different keys. This implementation is seen in unordered_set & unordered_map in C++, HashSet & HashMap in java, dict in python etc.

Compiler Operation:

The keywords of a programming language are processed differently than other identifiers. To differentiate between the keywords of a programming language(if, else, for, return etc.) and other identifiers and to successfully compile the program, the compiler stores all these keywords in a set which is implemented using a hash table.

Rabin-Karp Algorithm:

One of the most famous applications of hashing is the Rabin-Karp algorithm. This is basically a string-searching algorithm which uses hashing to find any one set of patterns in a string. A practical application of this algorithm is detecting plagiarism.

Linking File name and path together:

When moving through files on our local system, we observe two very crucial components of a file i.e. file_name and file_path. In order to store the correspondence between file_name and file_path the system uses a map(file_name, file_path)which is implemented using a hash table.