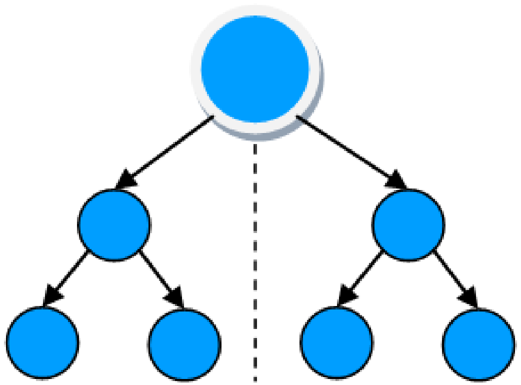
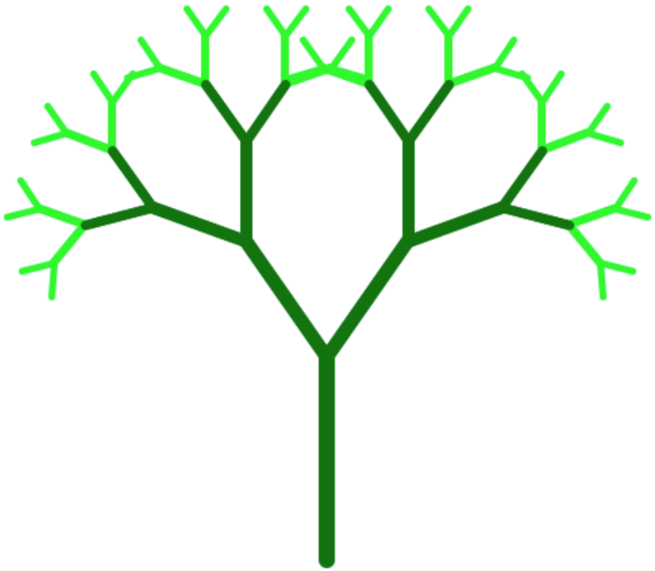
Lecture 4: **Binary Trees**

HERALD COLLEGE
KATHMANDU





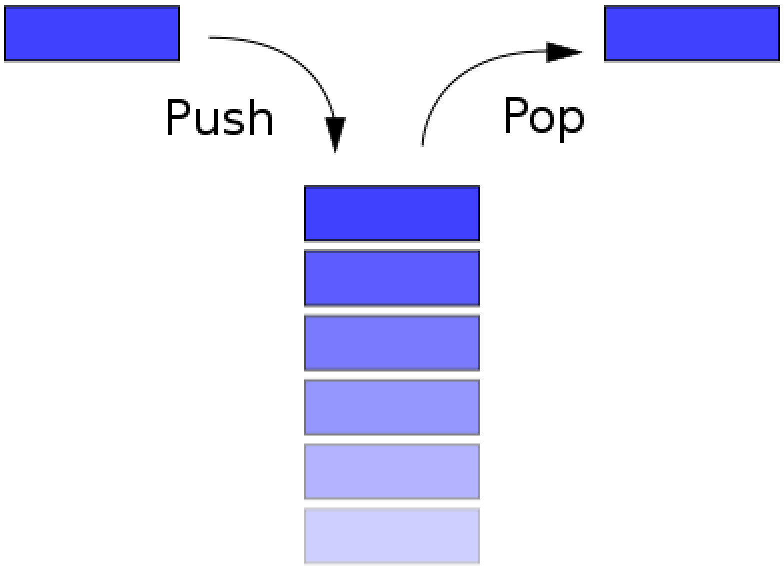
Objectives

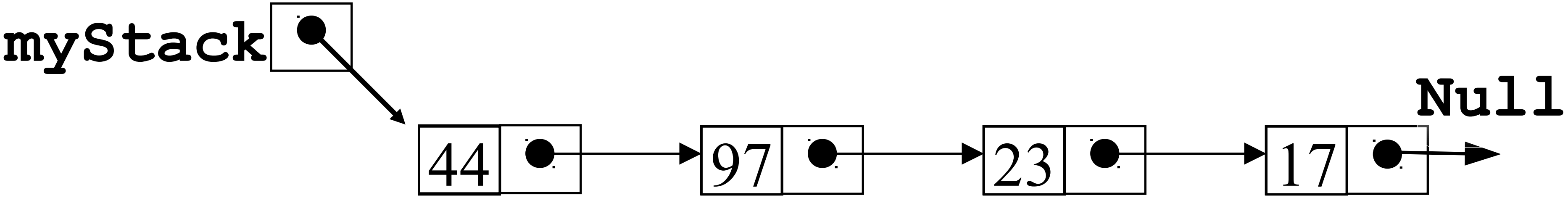
- Linear VS Non-linear Data Structures
- Binary Trees
- Binary Search Trees
- Operations
- Applications

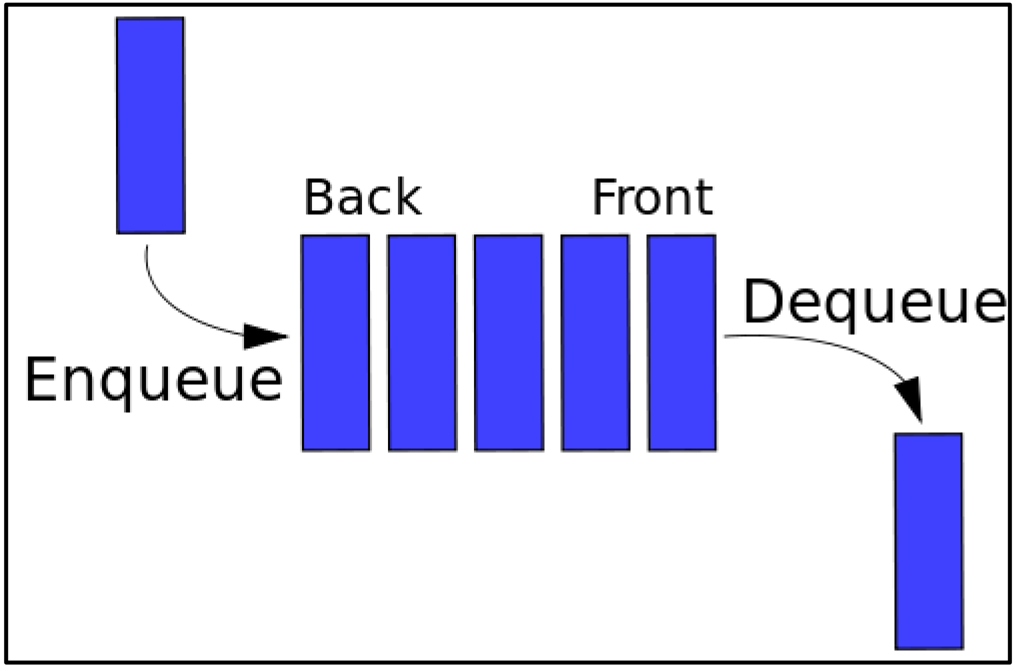
Agenda

- Binary Trees
- Binary Search Trees
- Traversal, Search
- Operations
- Depth First Search
- Breadth First Search
- Application

0	1	2	3	4	5	6	7	8	9
91	92	99	93	94	95	44	97	23	17




myStack
44
97
23
17
Null


Back
Front
Enqueue
Dequeue

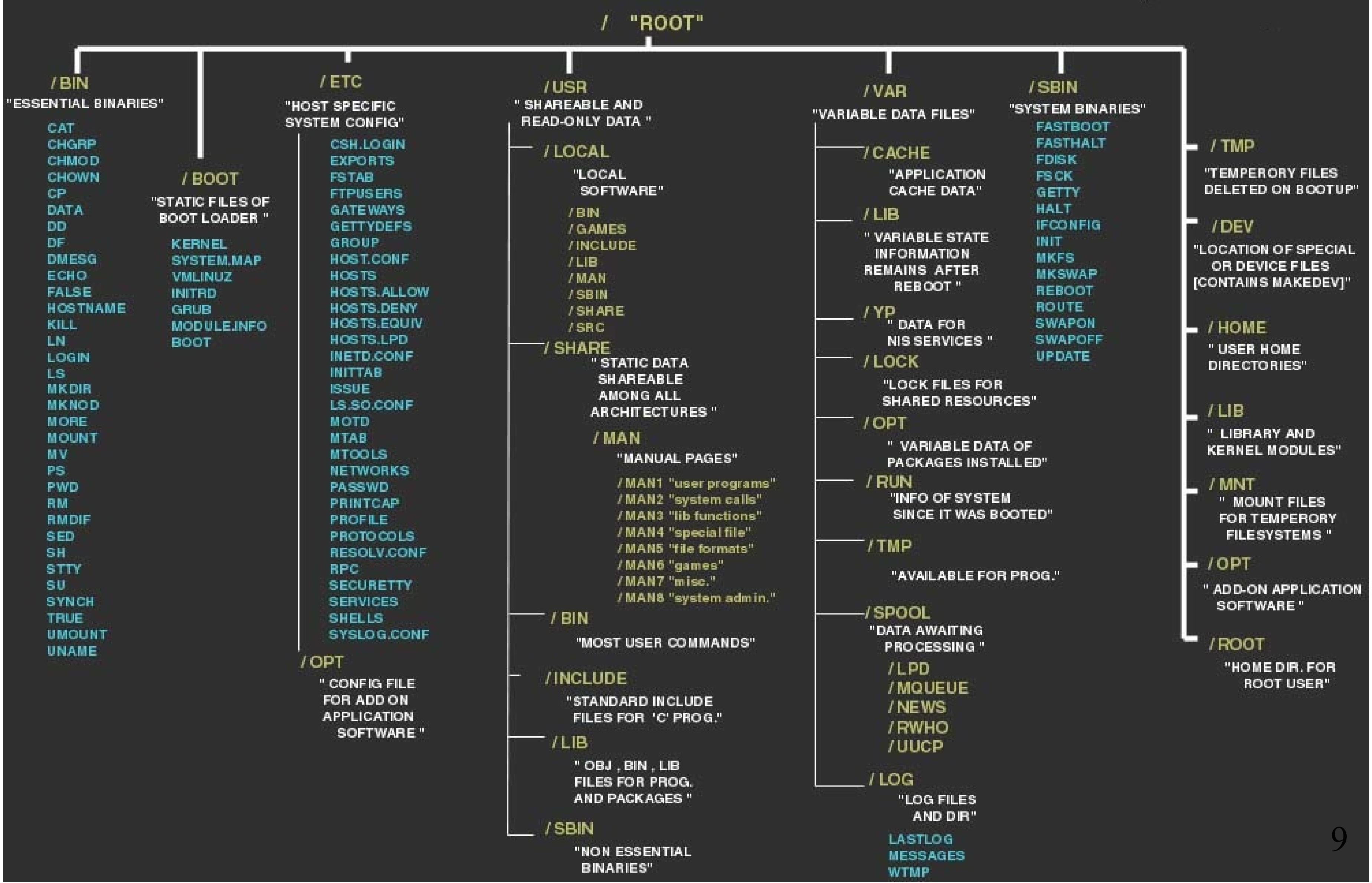
Linear DS	**Non-Linear DS**
Every item is related to its previous & next item	Every item is attached with many other items
Data is arranged in linear sequence	Data is not arranged in sequence
Data items can be traversed in a single run	Data items cannot be traversed in a single run
Array, Stack, Queue Linked List	Trees, Graphs
Implementation is easy	Implementation is difficult

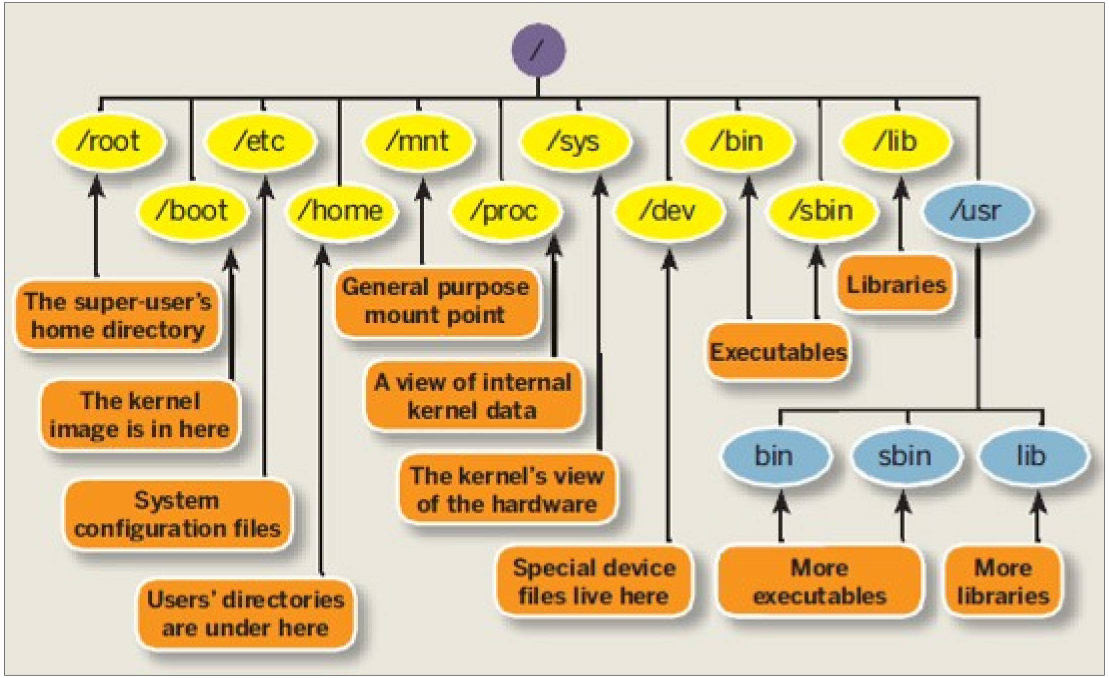
Data Structure Selection

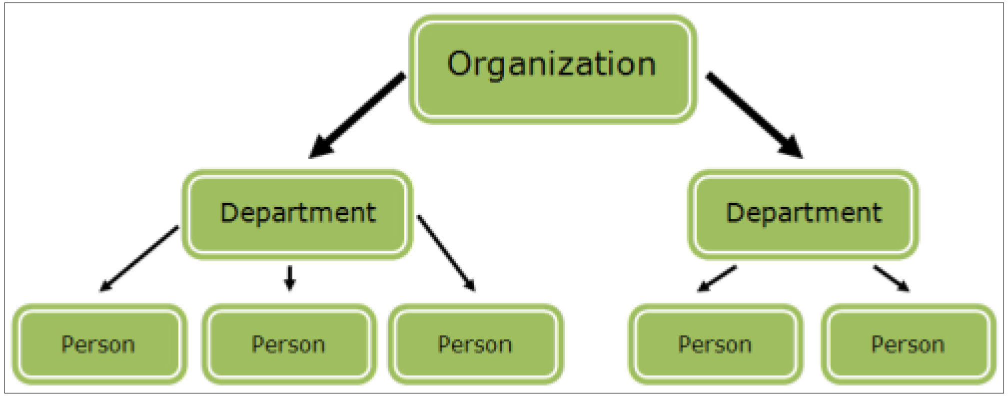
- What needs to be stored?
- Cost of operation
- Memory Usage
- Ease of Implementation

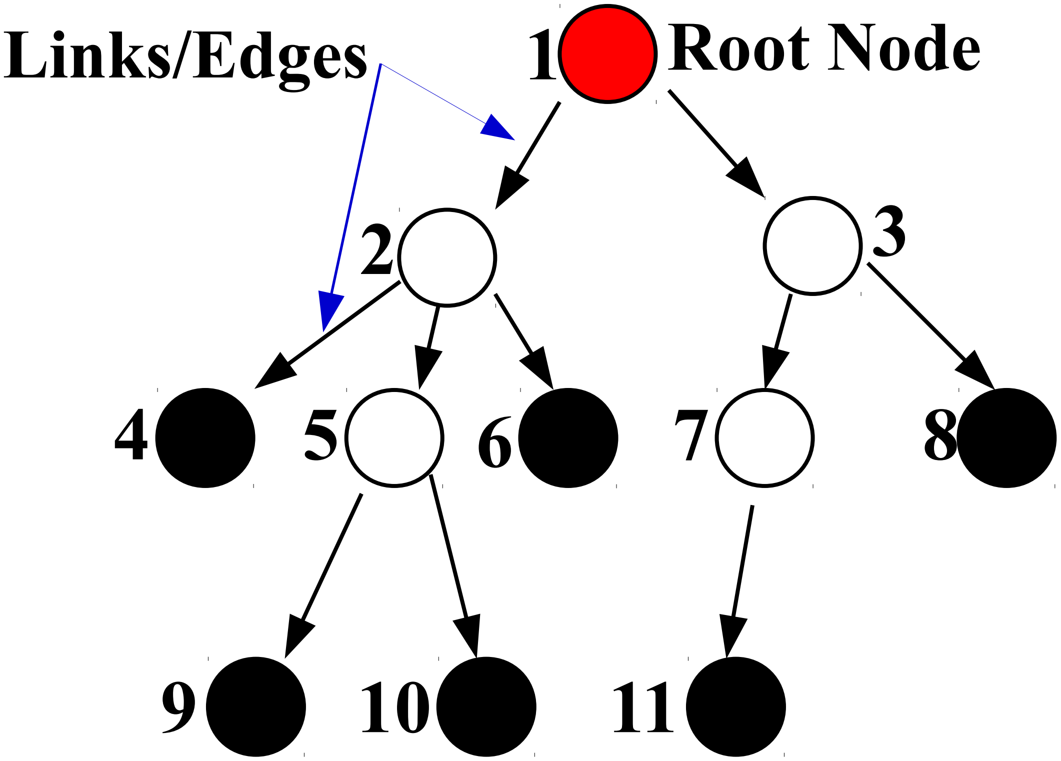
Trees

- Trees are useful for hierarchical data
  - ✔ A way to organize data if they are naturally hierarchical
  - ✔ A collection of entities called nodes linked together to simulate a hierarchy


/ "ROOT"
/ BIN
"ESSENTIAL BINARIES"
CAT
CHGRP
CHMOD
CHOWN
CP
DATA
DD
DF
DMESG
ECHO
FALSE
HOSTNAME
KILL
LN
LOGIN
LS
MKDIR
MKNOD
MORE
MOUNT
MV
PS
PWD
RM
RMDIF
SED
SH
STTY
SU
SYNCH
TRUE
UMOUNT
UNAME
/ BOOT
"STATIC FILES OF BOOT LOADER "
KERNEL
SYSTEM.MAP
VMLINUZ
INITRD
GRUB
MODULE.INFO
BOOT
/ ETC
"HOST SPECIFIC SYSTEM CONFIG"
CSH.LOGIN
EXPORTS
FSTAB
FTPUSERS
GATEWAYS
GETTYDEFS
GROUP
HOST.CONF
HOSTS
HOSTS.ALLOW
HOSTS.DENY
HOSTS.EQUIV
HOSTS.LPD
INETD.CONF
INITTAB
ISSUE
LS.SO.CONF
MOTD
MTAB
MTOOLS
NETWORKS
PASSWD
PRINTCAP
PROFILE
PROTOCOLS
RESOLV.CONF
RPC
SECURETTY
SERVICES
SHELLS
SYSLOG.CONF
/ OPT
" CONFIG FILE FOR ADD ON APPLICATION SOFTWARE "
/ USR
" SHAREABLE AND READ-ONLY DATA "
/ LOCAL
"LOCAL SOFTWARE"
/ BIN
/ GAMES
/ INCLUDE
/ LIB
/ MAN
/ SBIN
/ SHARE
/ SRC
/ SHARE
" STATIC DATA SHAREABLE AMONG ALL ARCHITECTURES "
/ MAN
"MANUAL PAGES"
/ MAN1 "user programs"
/ MAN2 "system calls"
/ MAN3 "lib functions"
/ MAN4 "special file"
/ MAN5 "file formats"
/ MAN6 "games"
/ MAN7 "misc."
/ MAN8 "system admin."
/ BIN
"MOST USER COMMANDS"
/ INCLUDE
"STANDARD INCLUDE FILES FOR 'C' PROG."
/ LIB
" OBJ , BIN , LIB FILES FOR PROG. AND PACKAGES "
/ SBIN
"NON ESSENTIAL BINARIES"
/ VAR
"VARIABLE DATA FILES"
/ CACHE
"APPLICATION CACHE DATA"
/ LIB
" VARIABLE STATE INFORMATION REMAINS AFTER REBOOT "
/ YP
" DATA FOR NIS SERVICES "
/ LOCK
"LOCK FILES FOR SHARED RESOURCES"
/ OPT
" VARIABLE DATA OF PACKAGES INSTALLED"
/ RUN
"INFO OF SYSTEM SINCE IT WAS BOOTED"
/ TMP
"AVAILABLE FOR PROG."
/ SPOOL
"DATA AWAITING PROCESSING "
/ LPD
/ MQUEUE
/ NEWS
/ RWHO
/ UUCP
/ LOG
"LOG FILES AND DIR"
LASTLOG
MESSAGES
WTMP
/ SBIN
"SYSTEM BINARIES"
FASTBOOT
FASTHALT
FDISK
FSCK
GETTY
HALT
IFCONFIG
INIT
MKFS
MKSWAP
REBOOT
ROUTE
SWAPON
SWAPOFF
UPDATE
/ TMP
"TEMPERORY FILES DELETED ON BOOTUP"
/ DEV
"LOCATION OF SPECIAL OR DEVICE FILES [CONTAINS MAKEDEV]"
/ HOME
" USER HOME DIRECTORIES"
/ LIB
" LIBRARY AND KERNEL MODULES"
/ MNT
" MOUNT FILES FOR TEMPERORY FILESYSTEMS "
/ OPT
" ADD-ON APPLICATION SOFTWARE "
/ ROOT
"HOME DIR. FOR ROOT USER"


/
/root
/boot
/etc
/home
/mnt
/proc
/sys
/dev
/bin
/sbin
/lib
/usr
The super-user's home directory
The kernel image is in here
System configuration files
Users' directories are under here
General purpose mount point
A view of internal kernel data
The kernel's view of the hardware
Special device files live here
Executables
Libraries
bin
sbin
lib
More executables
More libraries


Organization
Department
Department
Person
Person
Person
Person
Person

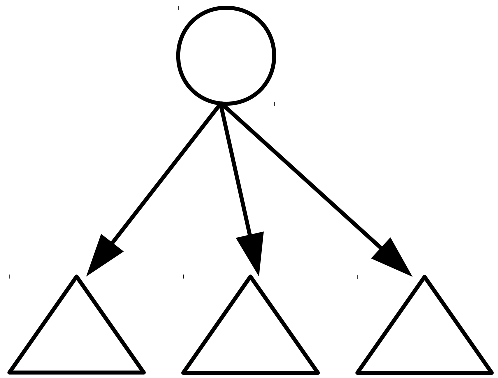


Except Leaf nodes: **Internal Nodes**

a) Root Node: **1**
b) Children of **1**: **2,3**
c) Sibling: {**4,5,6**} {**7,8**}…
d) Leaf Node: **4,6,8,9,10,11**
e) **1** is grandparent of **4,5,6**
f) **4** is grandchild of **1**
g) Ancestors of **10**: **1,2,5**
h) **10** is descendant of **5,2,1**
i) **Are 6,7 Siblings?** **(Cousins)**
j) **3** is uncle of **6**
k) Common ancestors of 4 & 9 **??**

Properties of Trees

- Recursive data structure
  - ✔ A tree is composed of smaller trees (**subtrees**) & leaf nodes
- If there are **n** nodes, there will be exactly **n-1** edges
- There will be one incoming link for each node except root



Sub-Trees

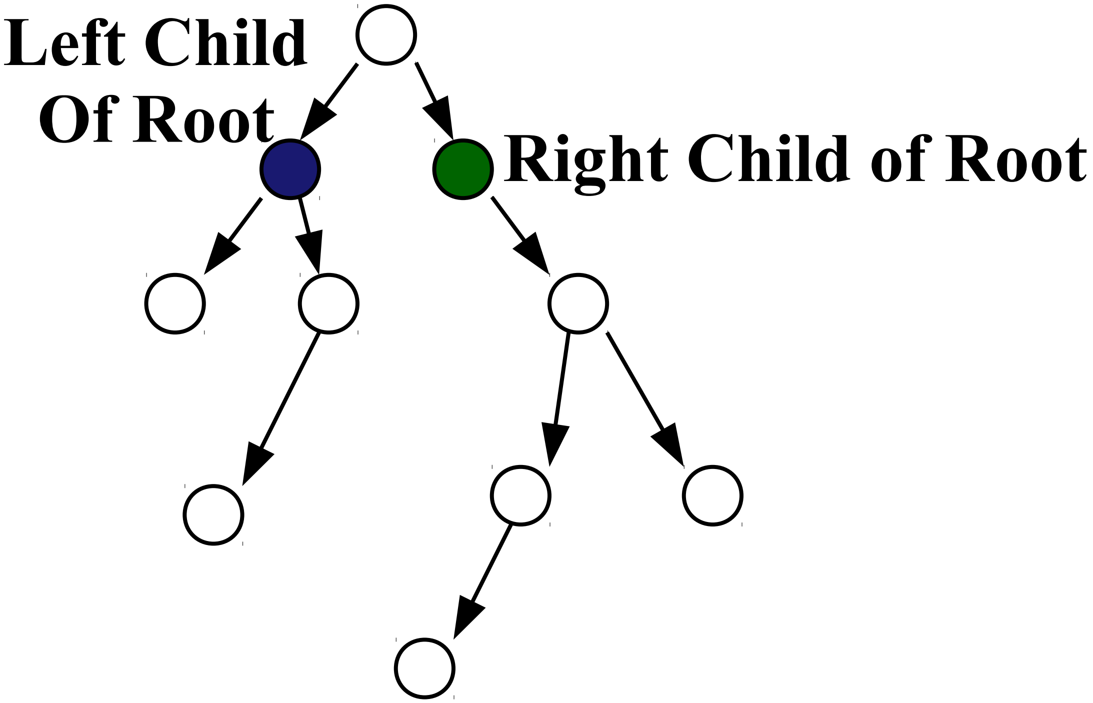
- Depth & Height
  - ✔ **Depth** of **n**th node = **no. of edges** in path from **Root** to **n**
  - ✔ **Height** of **n**th node = **no. of edges** in longest path from **n** to **Leaf**
  - ✔ **Depth** of **Root Node = 0**
  - ✔ **Height** of **tree = Height** of **Root Node**

Applications

- Storing naturally hierarchical data
  - ✔ File system on your disk drive
  - ✔ File & Folder hierarchy is naturally hierarchical data
- Organizing data, collection
- Dictionary
- Network Routing Algorithms

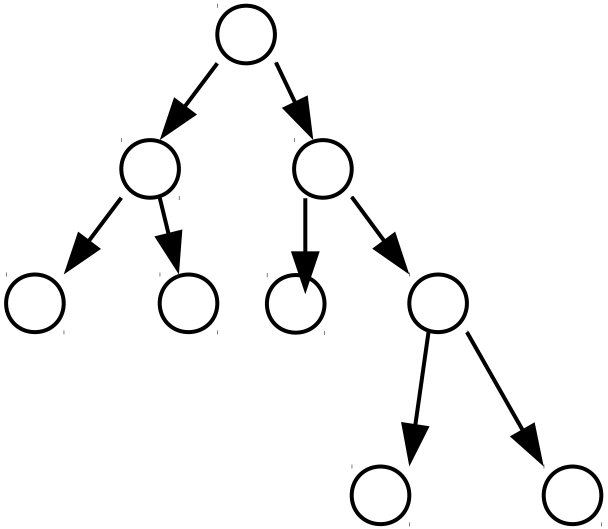
Binary Trees

- A tree in which each node can have at most 2 children

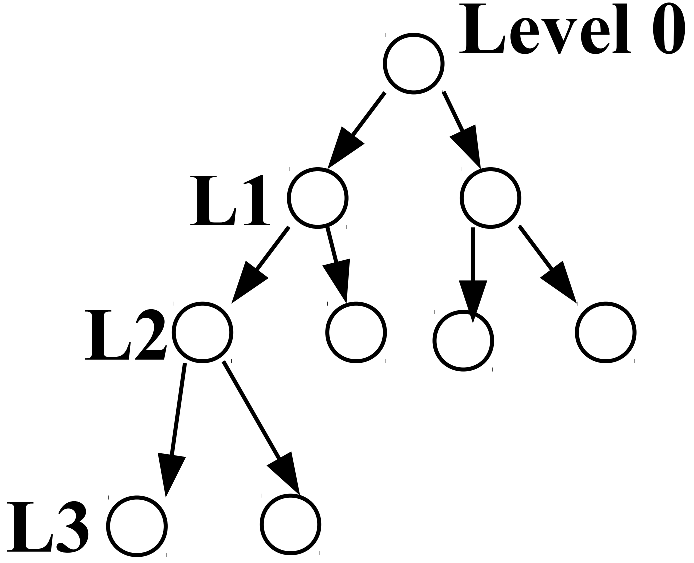


- If a tree has just a single node, then also its called a **BT**
- **Types of BT**
  - ✔ **Proper BT (Strict, Full, 2-Tree)**
  - ✔ **Complete BT**
  - ✔ **Perfect BT**
  - ✔ **A degenerate(Pathological) BT**
  - ✔ **Balanced BT**

- **Proper BT**
  - ✔ Each node can have either 2 or 0 children
  - ✔ Number of leaf nodes = no. of internal nodes + 1



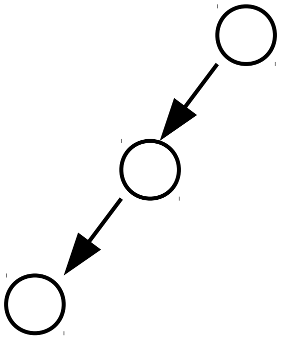
- **Complete BT**
  - ✔ All levels, except possibly last, is completely filled & all nodes are as far left as possible
  - ✔ Height of root node = Maximum depth of tree = height of the tree
  - ✔ Maximum no. of nodes at level **L** = $\mathbf{2^L}$



- **Perfect BT**
  - ✔ All internal nodes have two children & all leaves are at same level
  - ✔ **Are all proper BT are perfect?**
  - ✔ Maximum no. of nodes(**n**) with height h = $\mathbf{2^0+2^1+....+2^h}$
  - ✔ $\mathbf{n = 2^{h+1} -1 = 2^{\text{no. of levels}} -1}$
  - ✔ What will be the height of Perfect BT with n-nodes?

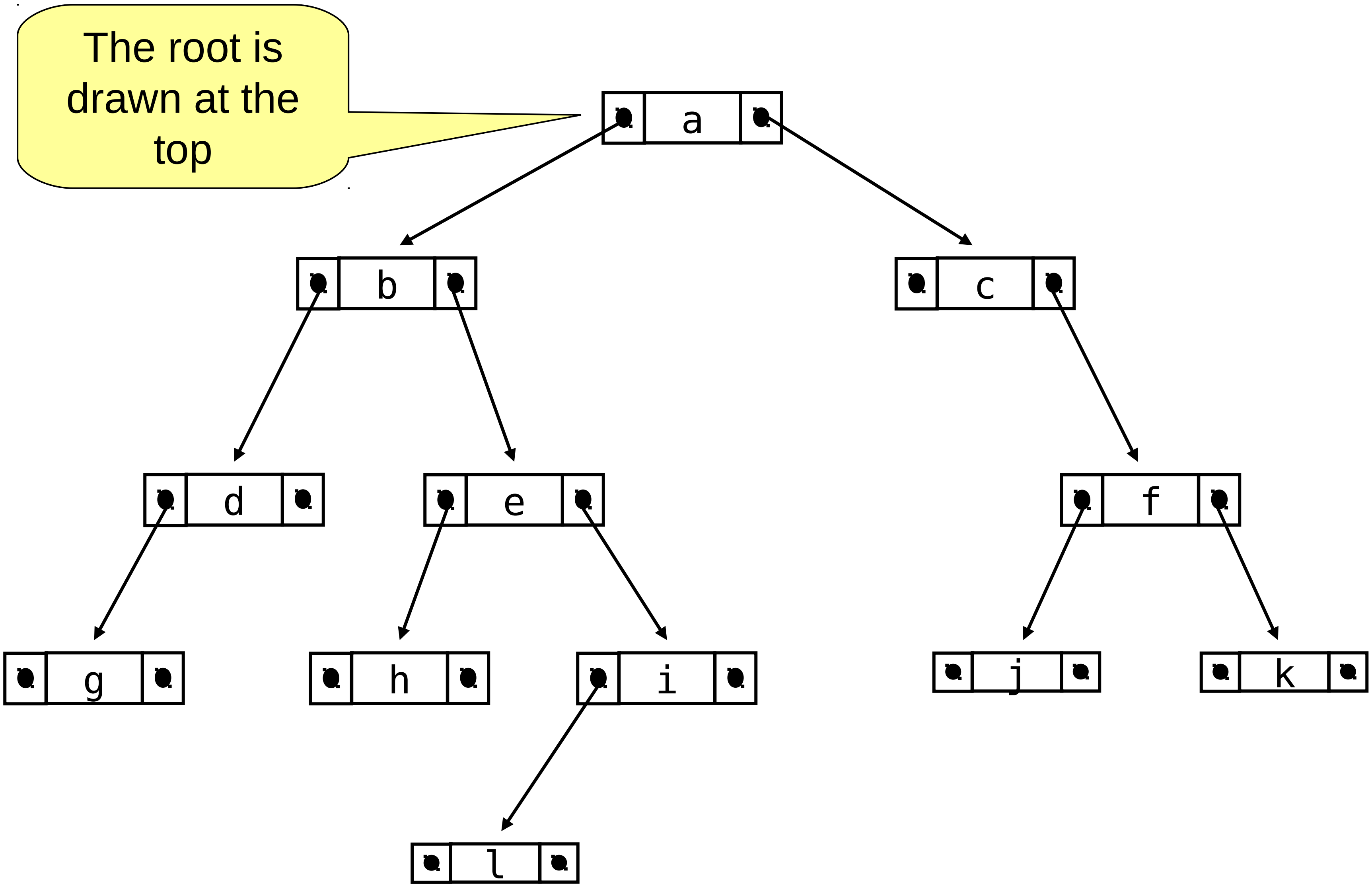
    $\mathbf{h = log(n+1)-1}$

- **A degenerate(or Pathological) BT**
  - ✔ A tree where every internal node has one child. These type of trees are performance wise same as linked-list.
  - ✔ **Maximum height = n-1**



- **Balanced BT**
  - ✔ A binary tree in which difference between height of left & right subtree for every node is not more than k (mostly 1)
  - ✔ **Difference = | $\text{Height}_{\text{left}}$ – $\text{Height}_{\text{right}}$ |**
  - ✔ **Height of an empty tree = -1**
  - ✔ **Height of a tree with just one node = 0**

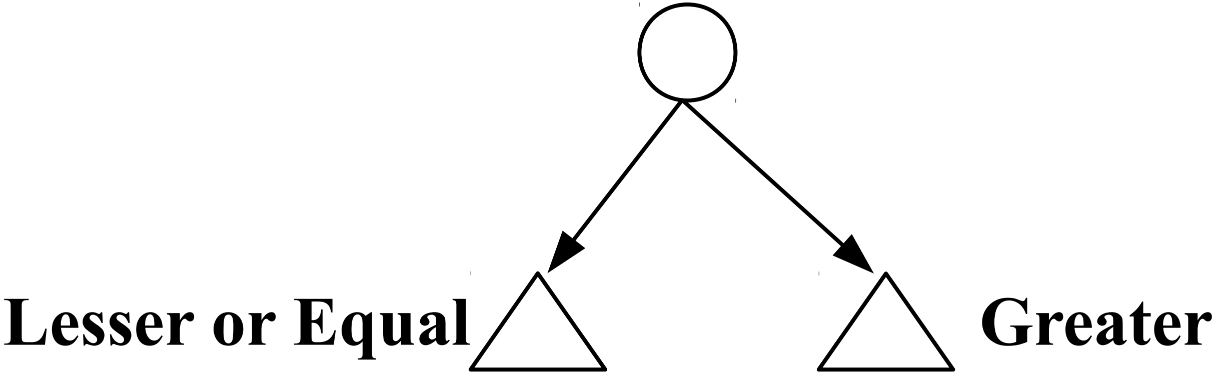
- **Implementation of BT**
  - ✔ Dynamically created nodes (Linked List)
  - ✔ Arrays(In case of Complete BT)
    - **For node at index i**
    - Left-child-index = 2i+1
    - Right-child-index = 2i+2

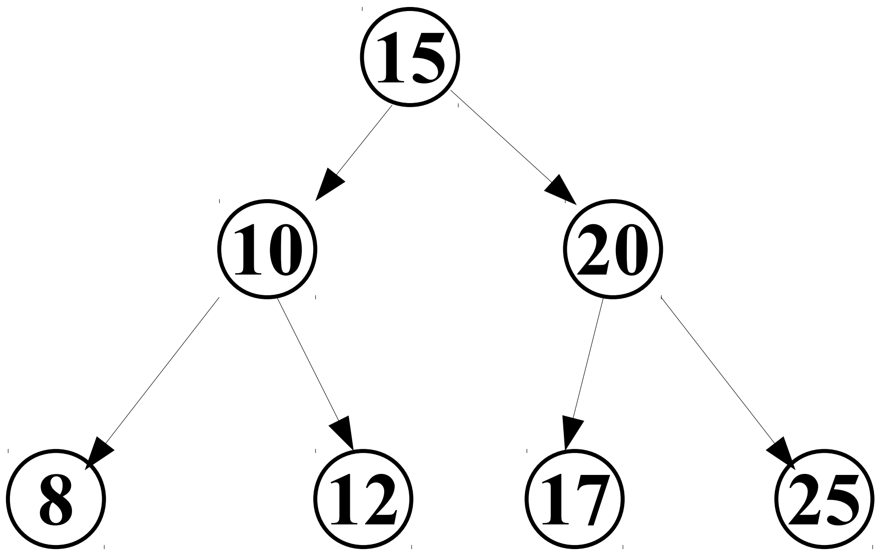

The root is drawn at the top
a
b
c
d
e
f
g
h
i
j
k
l

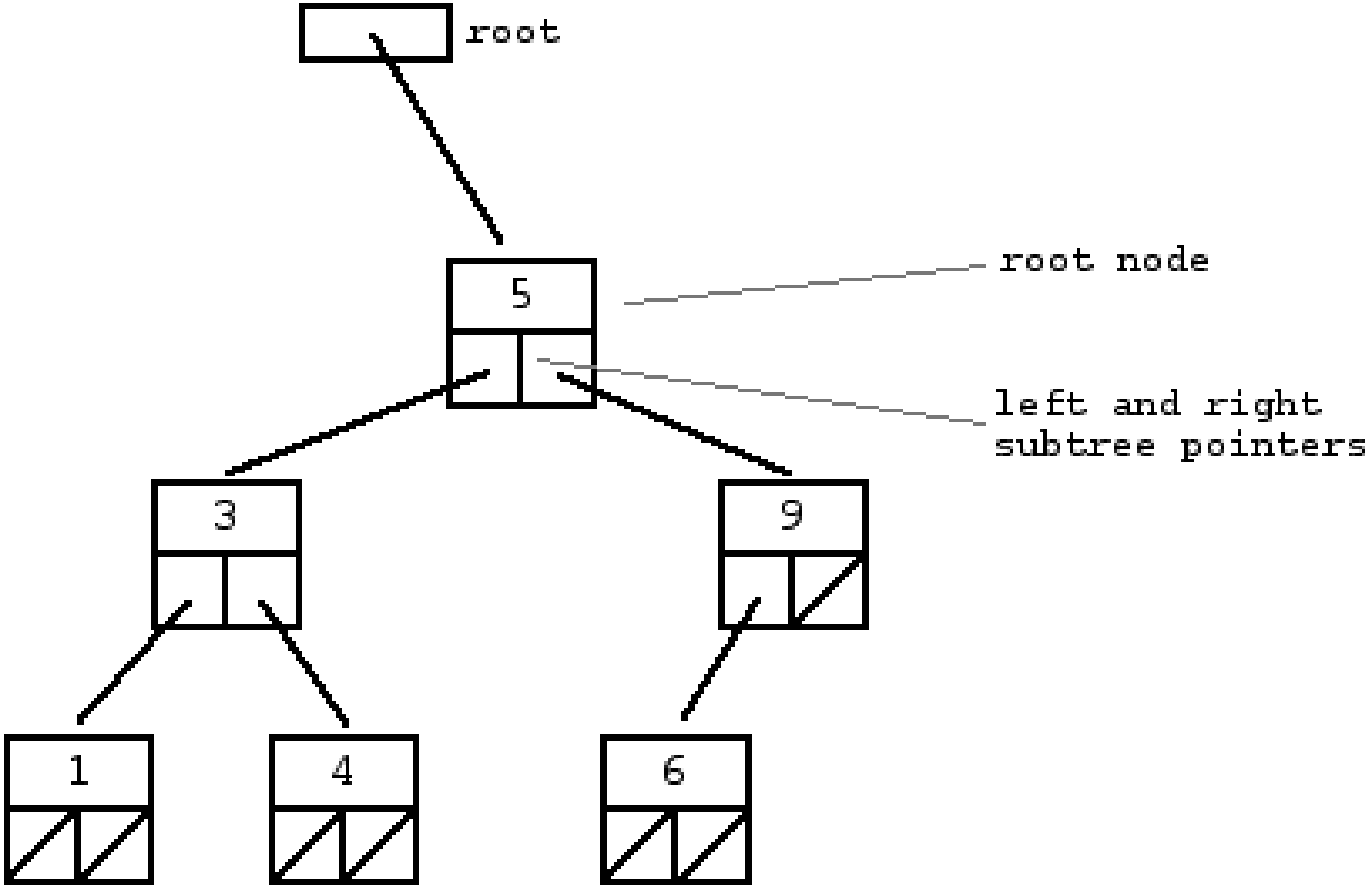
- **Binary Search Tree**

Operations	Array Unsorted	Linked List	Array Sorted	BST Balanced
Search(x)	O(n)	O(n)	O(logn)	O(logn)
Insert(x)	O(1)	O(1)	O(n)	O(logn)
Remove(x)	O(n)	O(n)	O(n)	O(logn)

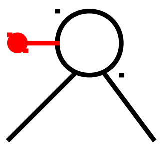
- **Binary Search Tree**
  - ✔ Binary search tree or Ordered binary tree where the nodes are arranged in order
  - ✔ For each node, all elements in its left subtree are less-or-equal to the node (<=), and all the elements in its right subtree are greater than the node (>).



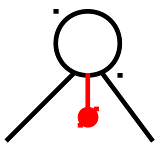

15
10
20
8
12
17
25


root
5
root node
left and right
subtree pointers
3
9
1
4
6

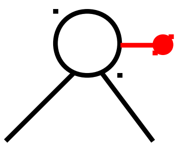
- **Tree Traversal**
  - ✔ Binary Tree consists of a root, a left subtree, and a right subtree
  - ✔ To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once (Depth First Search)
  - ✔ **`<Root><Left><Right> : Pre-Order`**
  - ✔ **`<Left><Root><Right> : In-Order`**
  - ✔ **`<Left><Right><Root>: Post-Order`**



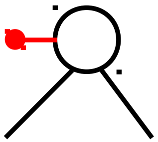
preorder



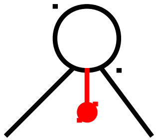
inorder



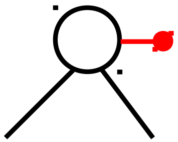
postorder



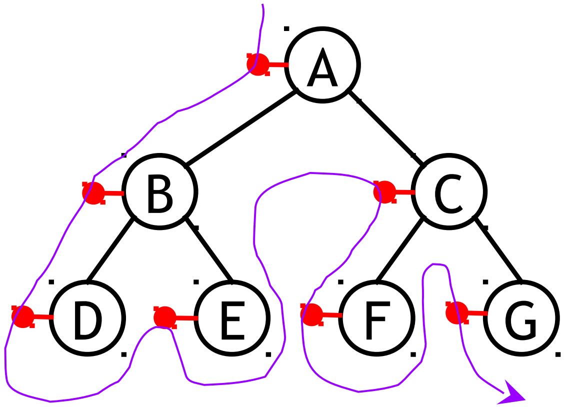
preorder



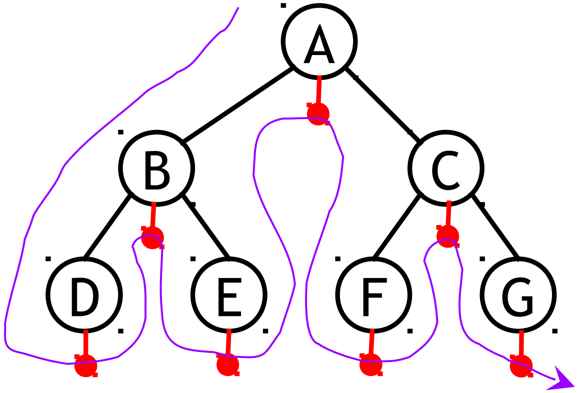
inorder



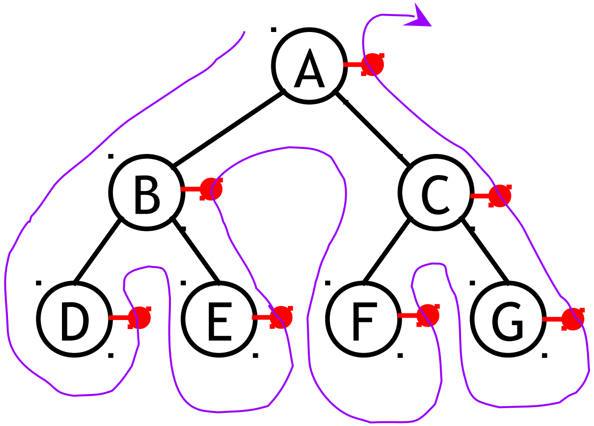
postorder



A B D E C F G



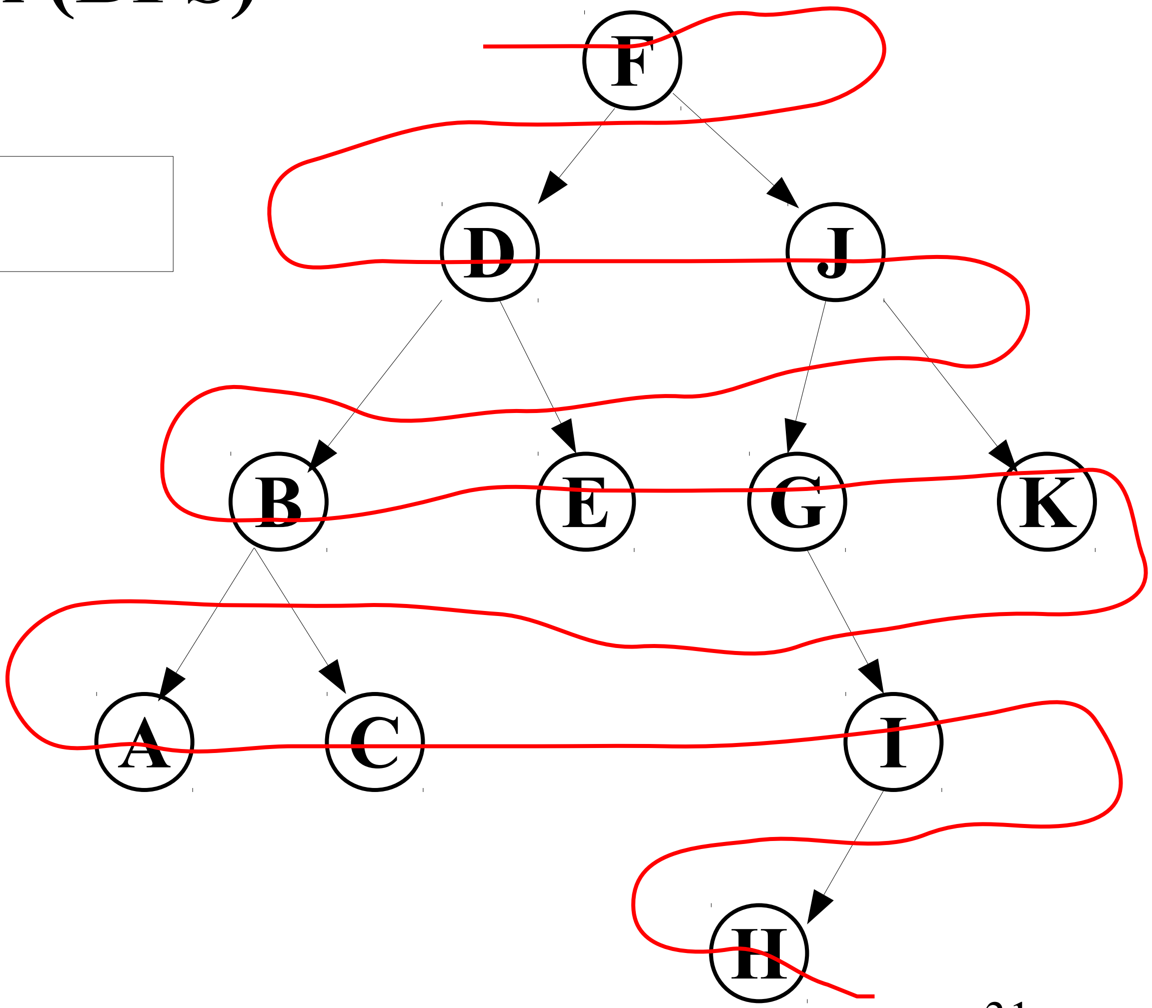
D B E A F C G



D E B F G C A

- **Level Order Traversal (BFS)**

FDJBEGKACIH



Full Binary Tree Theorem

- The number of leaves in a non-empty full binary tree is one more than the number of internal nodes
  - ✔ **No. of leaf nodes = No. of internal nodes + 1**
  - ✔ Relevant since it helps us calculate space requirements
- **Proof** by Mathematical Induction
  - ✔ **Base Case:** A full binary tree with **0** internal node has **1** leaf node

- **Proof** by Mathematical Induction
  - ✔ **Base Case:** A full binary tree with **0** internal node has **1** leaf node
  - ✔ **Induction Hypothesis:** Assume any full binary tree **T** containing **n − 1** internal nodes has **n** leaves
  - ✔ **Induction Step:** Given a full tree **T** with **n − 1** internal nodes (⇒ **n leaves**), add two leaf nodes as children of one of its leaves ⇒ obtain a tree **T'** having **n** internal nodes and **n + 1** leaves

Full Binary Tree Theorem Corollary

- The number of empty subtrees in a non-empty binary tree is **one more than** the number of nodes in the tree
- **Proof**
  - ✔ Replace all empty subtrees with a leaf node. This is a full binary tree, having

    leaves = empty subtrees of original tree

Binary Tree Node ADT

```
interface BinNode { // ADT for binary tree nodes
// Return and set the element value
public Object element();
public Object setElement(Object v);
// Return and set the left child
public BinNode left();
public BinNode setLeft(BinNode p);
// Return and set the right child
public BinNode right();
public BinNode setRight(BinNode p);
// Return true if this is a leaf node
public boolean isLeaf(); }
```

Traversals

- Any process for visiting the nodes in some order is called a **traversal**
- Depth First Search

- Depth First Search
  - ✔ Pre-Order (Root, Left, Right)
  - ✔ In-Order (Left, Root, Right)
  - ✔ Post-Order (Left, Right, Root)

✔ Reverse Pre-Order (Root, Right, Left)

✔ Reverse In-Order (Right, Root, Left)

✔ Reverse Post-Order (Right, Left, Root)

DFS(Basic Pseudocode) O(n)

```
• Initialize an empty stack for storage of nodes, S.
• For each node n, define n.visited to be false.
• Push the root (first node to be visited) onto S.
• While S is not empty:
      Pop the first element in S, n.
      If n.visited = false, then:
          n.visited = true
          for each unvisited neighbor p of n:
              Push p into S.
End process when all nodes have been visited.
```

DFS(Pre-Order) O(n)

- **Create an empty stack S & push root node to S**
- **while S is not empty.**

  Pop an item from stack and print it
  Push right child of popped item to stack
  Push left child of popped item to stack

// Right child is pushed before left child to make sure that left subtree is processed first.

BFS O(n)

```
• BFS (T, s)
  //Where T is the Tree & s is the root node
• let Q be queue.
  //Inserting s in queue until all nodes marked
      Q.enqueue( s )
• mark s as visited
• while ( Q is not empty)
  //Removing node from queue,whose neighbor will be
  //visited now
      p =  Q.dequeue( )
• for all neighbours R of P in T,  if R: not visited
• Q.enqueue( R ) & mark R as visited
```

- Just before starting to explore level **n**, the queue holds all the nodes at level **n-1**
- In a typical tree, the number of nodes at each level increases *exponentially* with the depth
- Memory requirements may be infeasible
- There is *no* "recursive" breadth-first search equivalent to recursive depth-first search

Heaps

A *(binary) heap* data structure is an array object that can be viewed as a complete binary tree

- Each node of the tree corresponds to an element of the array that stores the value in the node
- Max Heap: **A[parent(i)] A[i]**
  - ✔ *The root of any sub-tree holds the **greatest** value in the sub-tree*
- Min Heap: **A[parent(i)] ≤ A[i]**
  - ✔ *The root of any sub-tree holds the least value in that sub-tree*

- **Operations**
  - ✔ **getMini():** It returns the root element of Min Heap. **O(1)**
  - ✔ **extractMin():** Removes the minimum element from Min Heap. **O(Logn)**
  - ✔ **insert():** Inserting a new key takes **O(Logn)** time. We add a new key at the end of the tree. If new key is greater than its parent, then we don't need to do anything. Otherwise…!
  - ✔ **decreaseKey():** Decreases value of key. **O(Logn)** If the decreases key value of a node is greater than parent of the node, then we don't need to do anything. Otherwise…!

- **Max-Heapify**
  - ✔ Given a tree that is a heap except for node **i**,
  - ✔ Max-Heapify function arranges node **i** and it’s subtrees to satisfy the heap property

```
MAX-HEAPIFY(A,i)
  l = LEFT(i)
  r = RIGHT(i)
  if l <= A.heapsize and A[l] > A[i]
     largest = l
  else
     largest = i
  if r <= A.heapsize and A[r] > A[largest]
     largest = r
  if largest != i
    exchange A[i] with A[largest]
  else MAX-HEAPIFY(A,largest)
```

O(h)

Priority Queue

- Extension of Queue with following properties:
  - ✔ Every item has a priority associated with it
  - ✔ An element with high priority is dequeued before an element with low priority
  - ✔ If two elements have the same priority, they are served according to their order in the queue
- Using a heap to implement a priority queue, we will always have the element of highest priority in the root node of the heap

- **Operations**
  - ✔ **getHighestPriority( ):** O(1)
  - ✔ **insert( ):** O(logn)
  - ✔ **deleteHighestPriority( ):** O(logn)

...?

Thank You