Introduction to Data Structures

Data Structures

- Data Structures A data structure is a scheme for organizing data in the memory of a computer. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs The way in which the data is organized affects the performance of a program for different tasks

Types of Data Structure

- There are basically two types of data structure
- Linear Data Structure: Stack, Queue,Linked List
- Non-Linear Data Structure. Tree And Graph

Stack

- Stack is a linear data structure which works on LIFO or FILO order i.e. First In Last Out or Last In First Out.
- In Stack element is always added at top of stack and also removed from top of the stack.
- Stack is useful in recursive function, function calling, mathematical expression, calculation, reversing the string etc.

Queue

- Queue is also a linear data structure which work on FIFO order i.e. First In First Out.
- In queue element is always added at rear of queue and removed from front of queue.
- Queue applications are in CPU scheduling, Disk Scheduling, IO Buffers, pipes, file input output.

Linked List

- A linked list is a linear collection of data elements, in which linear order is not given by their physical placement in memory.
- Elements may be added in front, end of list as well as middle of list.
- Linked list may use for dynamic implementation of stack and queue.

Trees

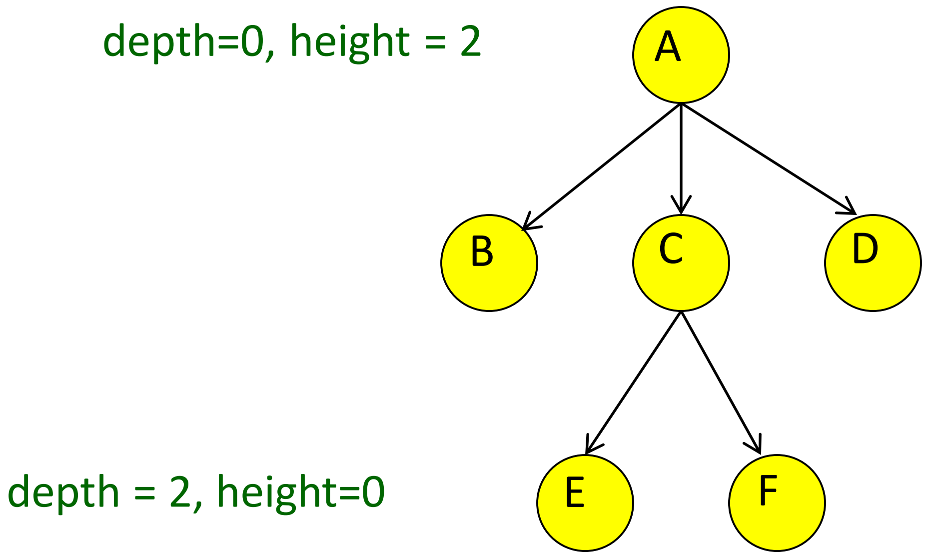
- A tree is a non linear data structure. a root value and subtrees of children with a parent node, represented as a set of linked nodes. Nodes can be added at any different node. Tree applications includes:-
- Manipulate hierarchical data.
- Make information easy to search (see tree traversal).
- Manipulate sorted lists of data.
- As a workflow for compositing digital images for visual effects.
- Router algorithms

Graphs

- A graph is a non linear data structure. A set of items connected by edges. Each item is called a vertex or node.
- Formally, a graph is a set of vertices and a binary relation between vertices, adjacency.
- Graph applications:- finding shortest routes, searching, social network connections, internet routing.

Trees

- Length of a path = number of edges
- Depth of a node N = length of path from root to N
- Height of node N = length of longest path from N to a leaf
- Depth and height of tree = height of root



Definition

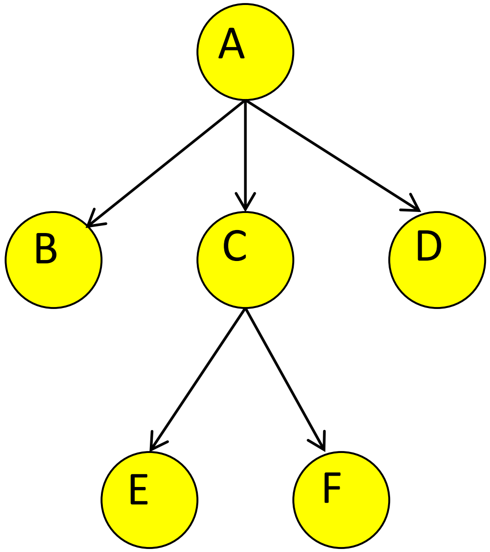
A tree is a set of nodes that is

a. an empty set of nodes, or

b. has one node called the root from which zero or more trees (sub trees) descend.

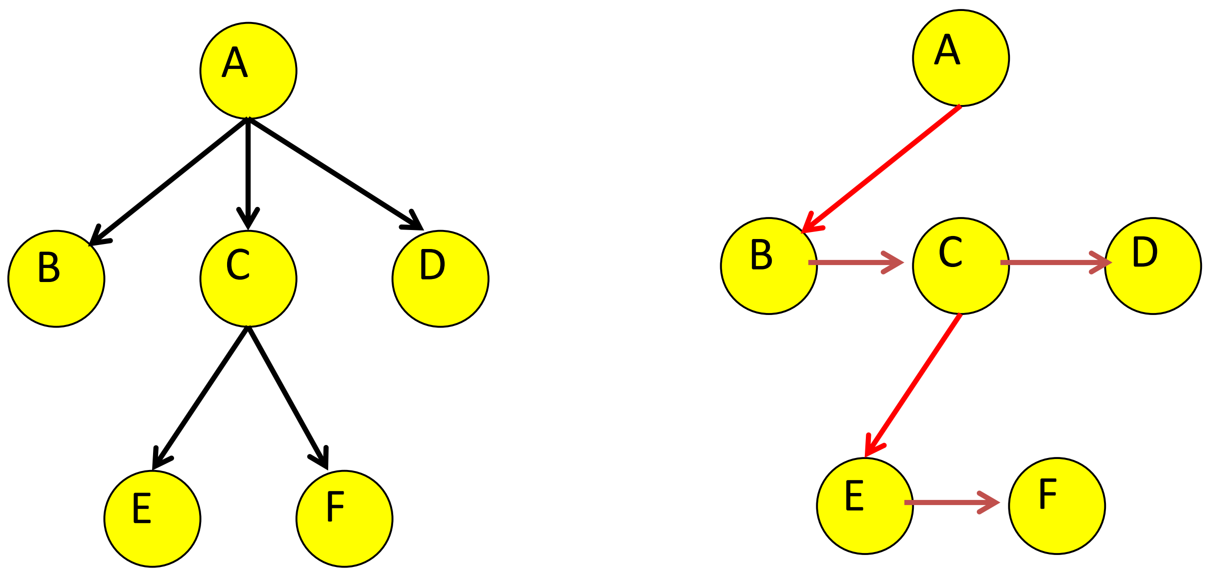
Implementation of Trees

- Obvious Pointer-Based Implementation: Node with value and pointers to children



1^{st} Child/Next Sibling Representation

- Each node has 2 pointers: one to its first child and one to next sibling

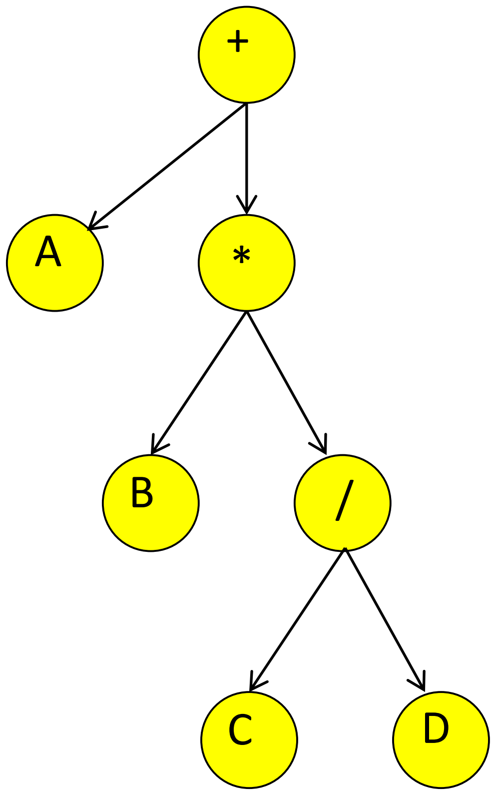


Application: Arithmetic Expression Trees

Example Arithmetic Expression:

A + (B * (C / D))

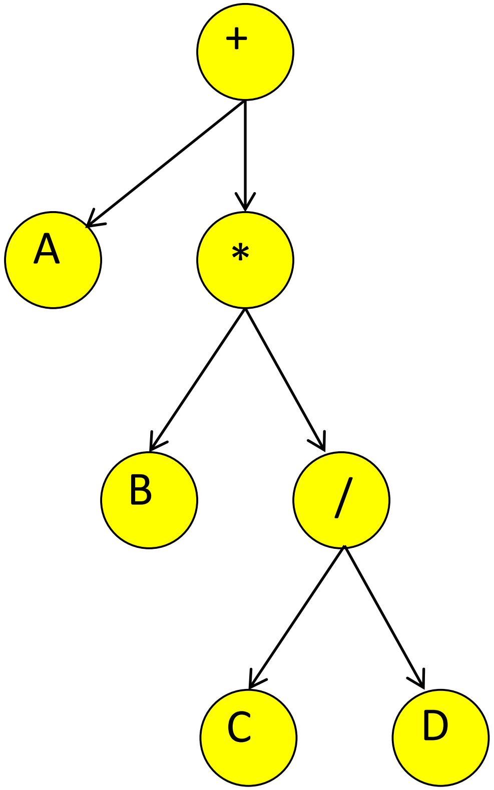
Tree for the above expression:



- Used in most compilers
- No parenthesis need – use tree structure
- Can speed up calculations e.g. replace / node with C/D if C and D are known
- Calculate by traversing tree (how?)

Traversing Trees

- Preorder: Root, then Children
  - + A * B / C D
- Postorder: Children, then Root
  - A B C D / * +
- Inorder: Left child, Root, Right child
  - A + B * C / D



Binary Trees

- Properties

  Notation:

  depth(tree) = MAX {depth(leaf)} = height(root)

  – max # of leaves = $2^{depth(tree)}$

  – max # of nodes = $2^{depth(tree)+1} - 1$

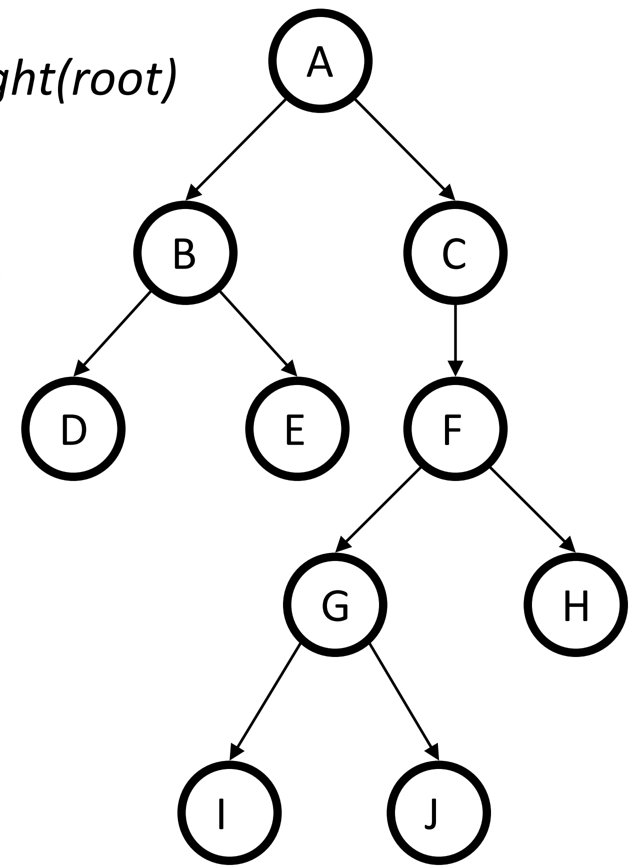
  – max depth = n-1

  – average depth for n nodes = $\sqrt{n}$

  (over all possible binary trees)

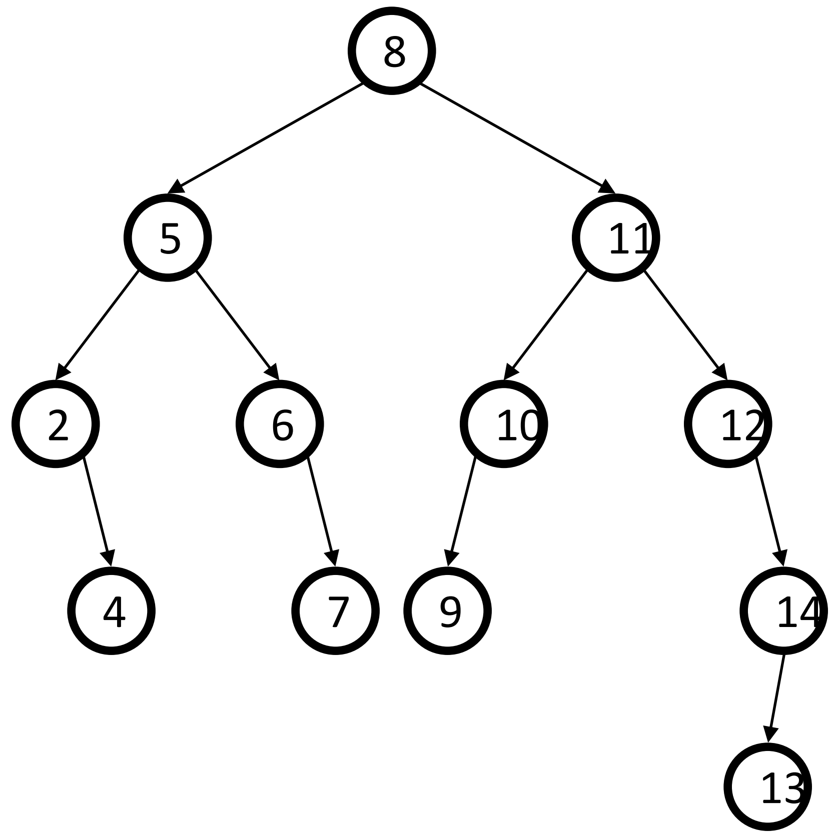
- Representation:

Data	
left pointer	right pointer

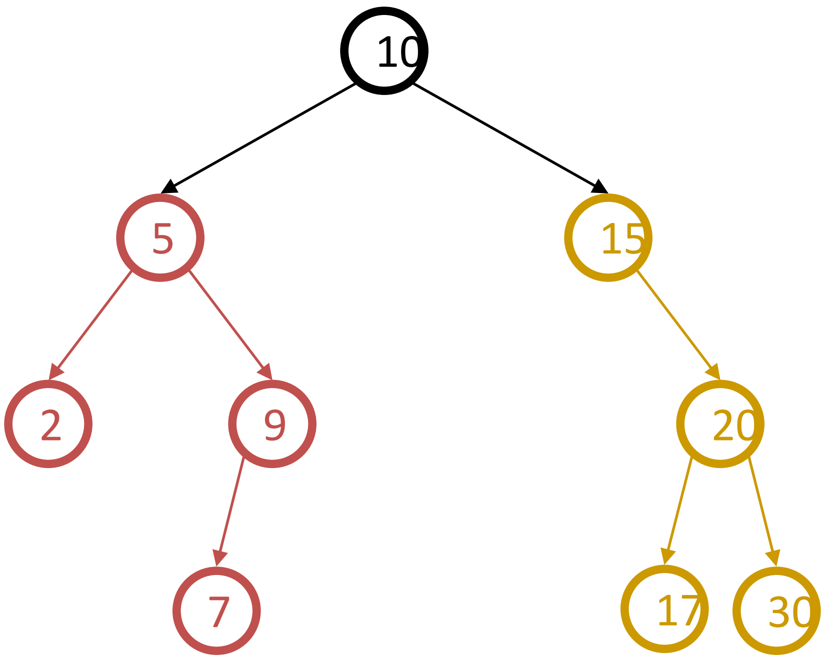


Binary Search Tree Dictionary Data Structure

- Search tree property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result:
    - easy to find any given key
    - inserts/deletes by changing links



In Order Listing



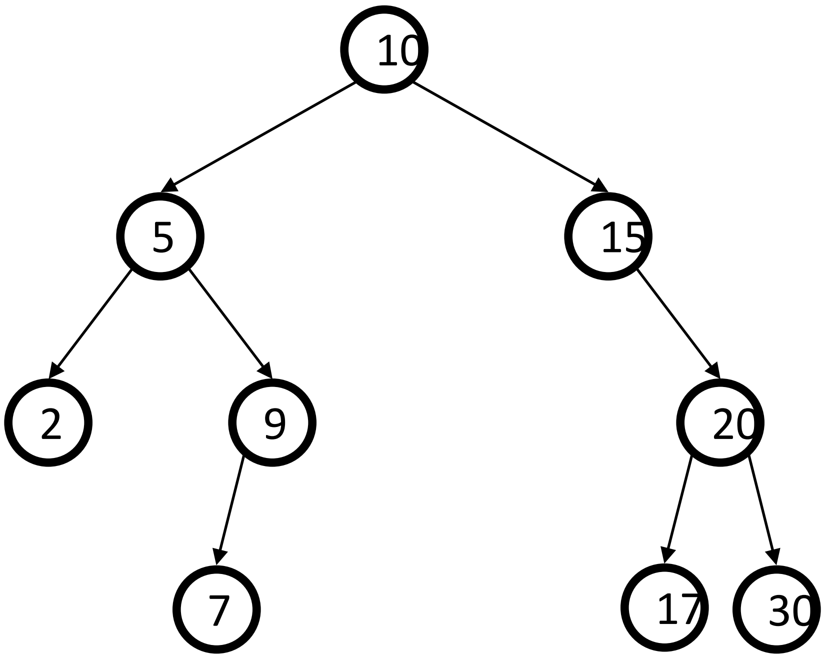
visit left subtree

visit node

visit right subtree

In order listing:
2→5→7→9→10→15→17→20→30

Finding a Node




```
Node find(Comparable x, Node
   root)
{
  if (root == NULL)
    return root;
  else if (x < root.key)
    return find(x,root.left);
  else if (x > root.key)
    return find(x, root.right);
  else
    return root;
}
```

Insert

Concept: proceed down tree as in Find; if new key not found, then insert a new node at last spot traversed

```
void insert(Comparable x,  Node root) {
  // Does not work for empty tree – when root is
  NULL
  if (x < root.key){
      if (root.left == NULL)
            root.left = new Node(x);
      else insert( x, root.left ); }
  else if (x > root.key){
      if (root.right == NULL)
            root.right = new Node(x);
      else insert( x, root.right ); } }
```

Time to Build a Tree

Suppose $a_1, a_2, \ldots, a_n$ are inserted into an initially empty BST:

1. $a_1, a_2, \ldots, a_n$ are in increasing order

2. $a_1, a_2, \ldots, a_n$ are in decreasing order

3. a_1 is the median of all, a_2 is the median of elements less than a_1, a_3 is the median of elements greater than a_1, etc.

4. data is randomly ordered

Analysis of BuildTree

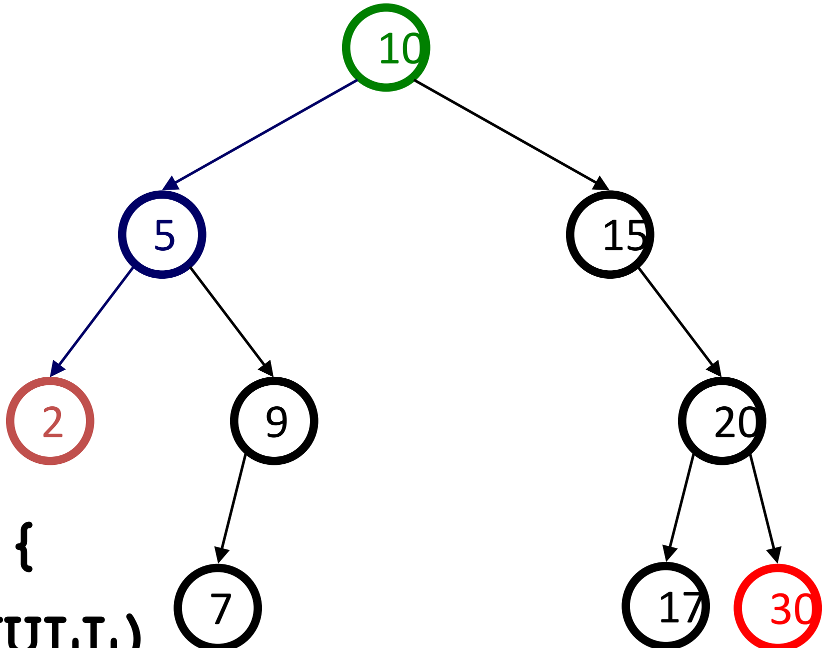
- Increasing / Decreasing: $\theta(n^2)$

  $1 + 2 + 3 + \ldots + n = \theta(n^2)$
- Medians – yields perfectly balanced tree

  $\theta(n \log n)$
- Average case assuming all input sequences are equally likely is $\theta(n \log n)$
  - equivalently: average depth of a node is *log n*

    Total time = sum of depths of nodes

FindMin/FindMax



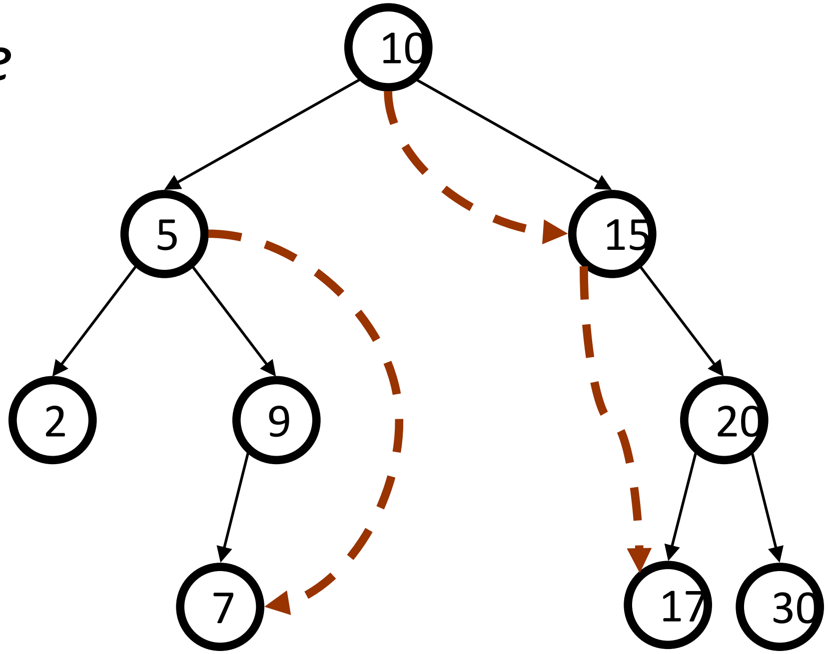

```
Node min(Node root) {
  if (root.left == NULL)
    return root;
  else
    return min(root.left);
  }
```

Successor

Find the next larger node
in this node's subtree.

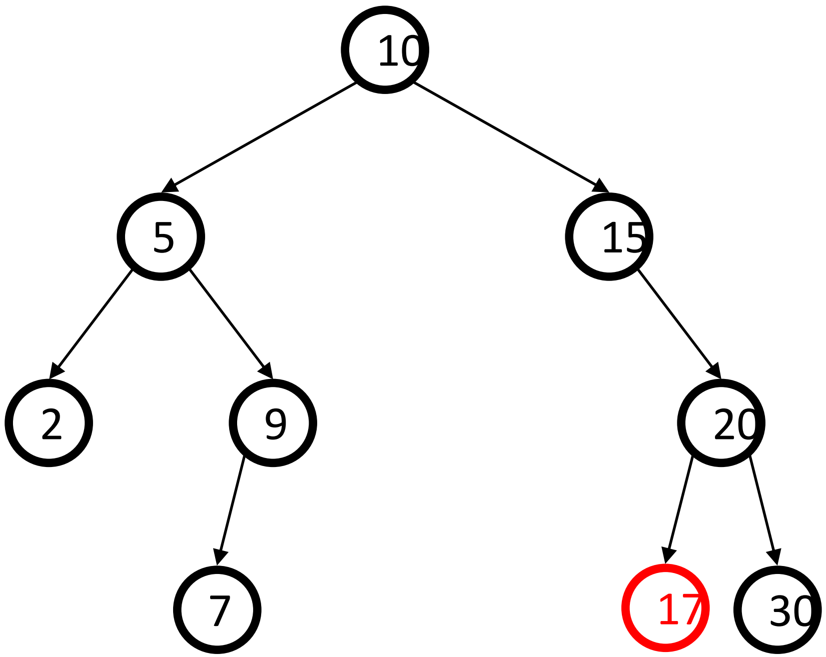
- *not next larger in entire tree*

```
Node succ(Node root) {
  if (root.right == NULL)
    return NULL;
  else
    return min(root.right);
}
```



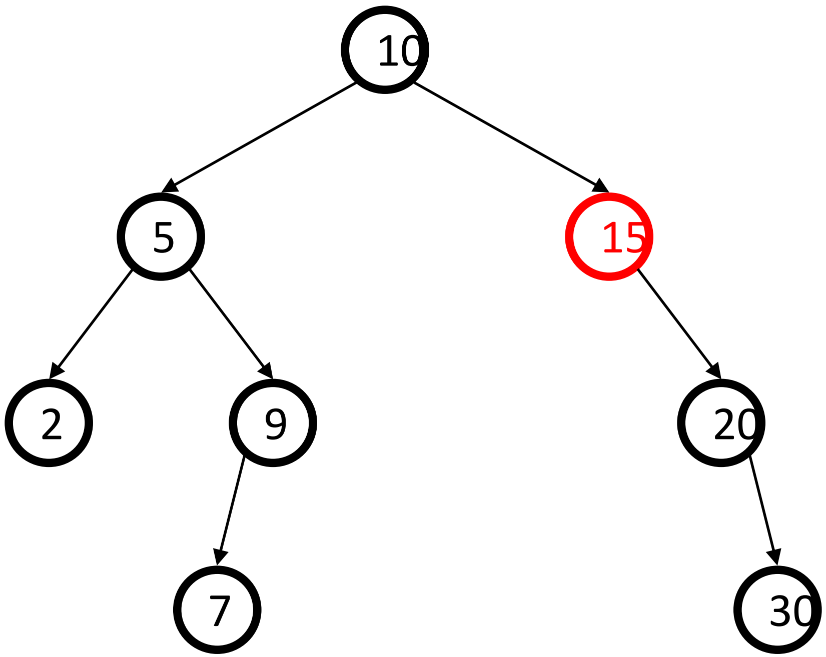
Deletion - Leaf Case

Delete(17)



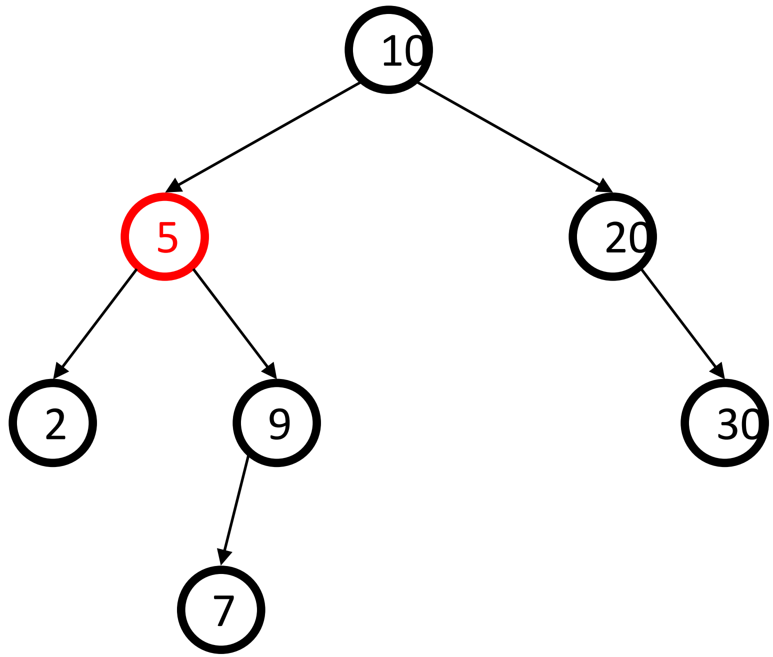
Deletion - One Child Case

Delete(15)



Deletion - Two Child Case

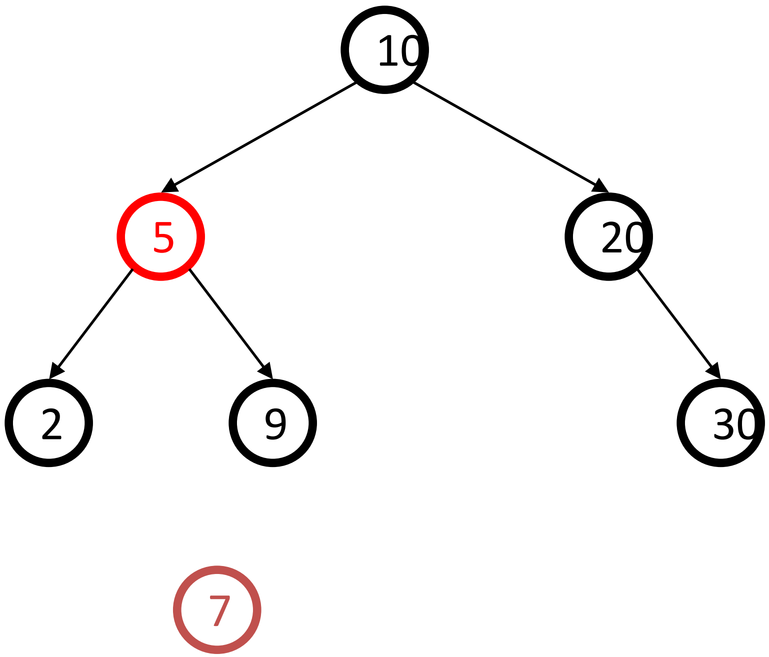
Delete(5)



replace node with value guaranteed to be between the left and right subtrees: the successor

Deletion - Two Child Case

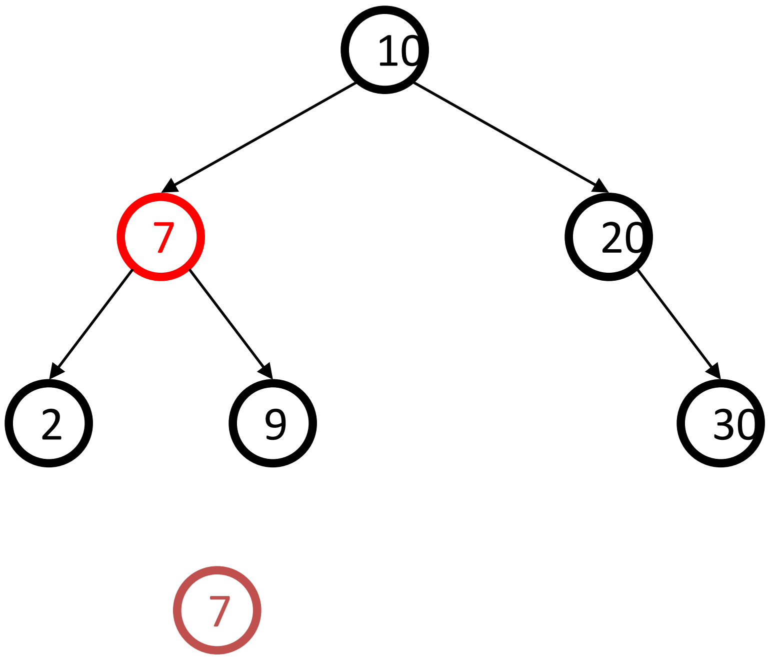
Delete(5)



always easy to delete the successor – always has either 0 or 1 children!

Deletion - Two Child Case

Delete(5)



Finally copy data value from deleted successor into original node