DATA STRUCTURES AND ALGORITHMS

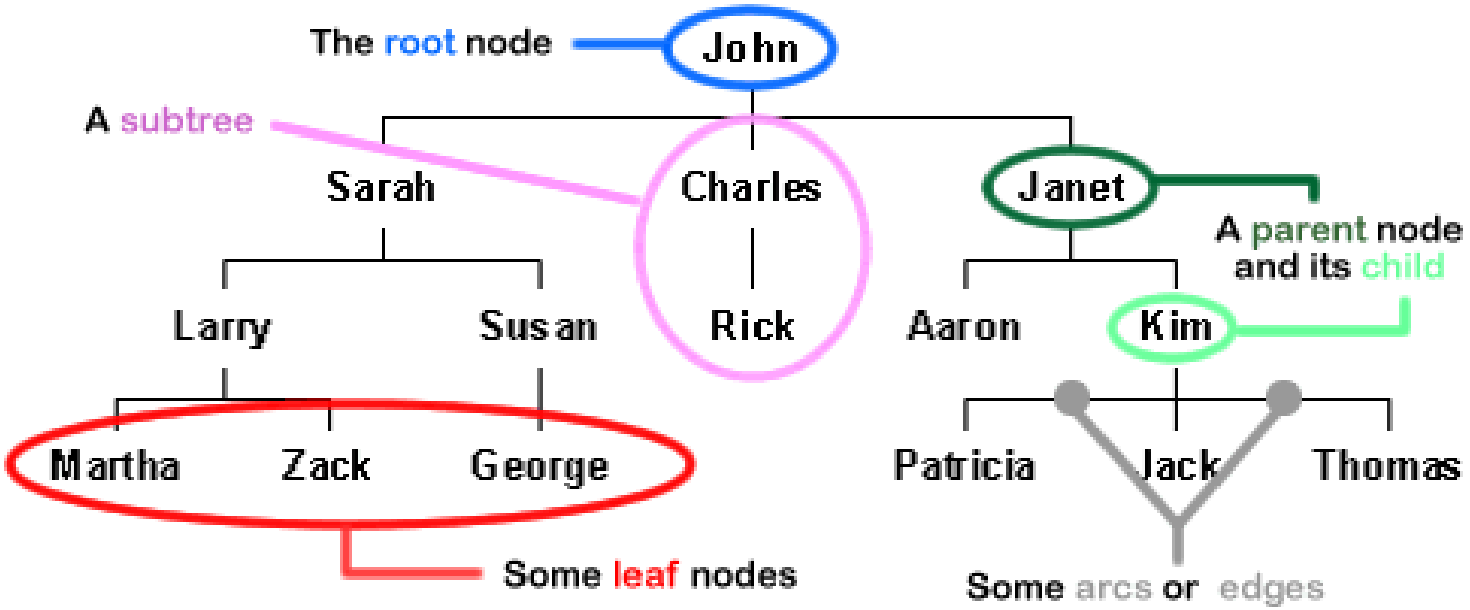
Hierarchical data structures:

Binary trees

Summary of the previous lecture

- Dynamic data structures
  - Lists
  - Stack
  - Queue
  - Deque
- Array – based implementation
- Linked list based implementation

Tree



Tree is non linear dynamic data structure that is very useful for searching.

Tree definitions

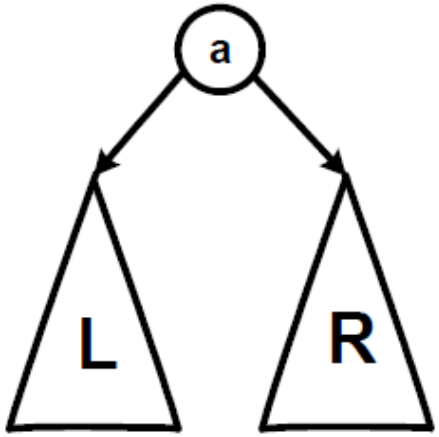
A tree is made up of a finite set of elements called **nodes**.

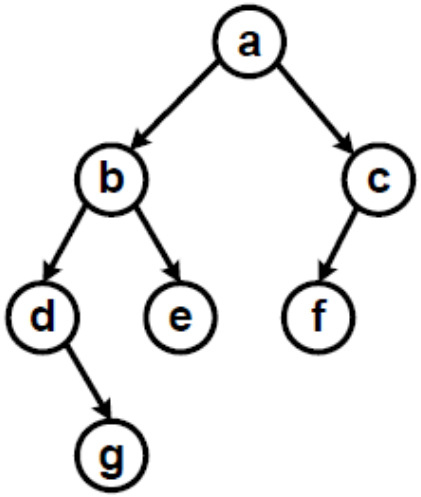
This set either is empty or consists of a node called the **root** together with two binary trees, called the **left and right subtrees**, which are disjoint from each other and from the root.
(*Disjoint mean have no nodes in common*.)

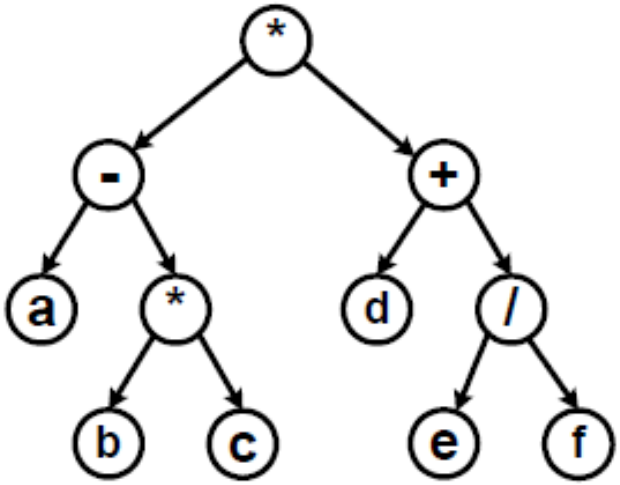
The roots of these subtrees are **children** of the root.

There is an edge from a node to each of its children, and a node is said to be the **parent of its children**.

Examples of the tree







$$(a - b * c) * (d + e/f)$$

Tree definitions

If n_1, n_2, ..., n_k is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $1 <= i < k$, then this sequence is called a **path** from n_1 to n_k.

The **length of the path** is k -1.

The **depth of a node** M in the tree is the **length** of the path from the root of the tree to M.

The **height** of a tree is one more than the depth of the deepest node in the tree.

All nodes of depth **d** are at level **d** in the tree.

The root is the only node at level 0, and its depth is 0.

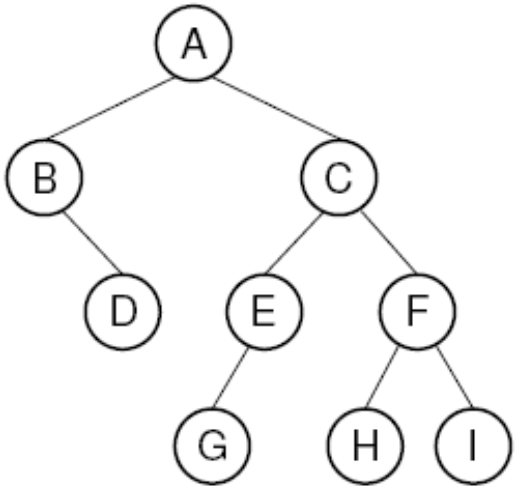
Tree definitions

A **leaf node** is any node that has no children.

An **internal node** is any node that has at least one non-empty child.

Binary trees have the restriction that nodes can't have more than two children.

Tree definitions



An example of binary tree. Node A is the root. Nodes B and C are A's children. Nodes B and D together form a subtree. Node B has two children: Its left child is the empty tree and its right child is D. Nodes A, C, and E are parents of G.
Nodes D, E, and F make up level 2 of the tree; node A is at level 0. The edges from A to C to E to G form a path of length 3. Nodes D, G, H, and I are leaves. Nodes A, B, C, E, and F are internal nodes. The depth of I is 3. The height of this tree is 4.

Binary tree (full, complete)

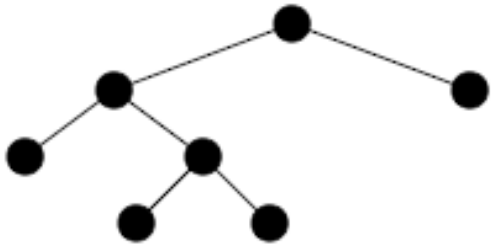
Each node in a **full binary tree** is either an internal node with exactly two non-empty children or a leaf.

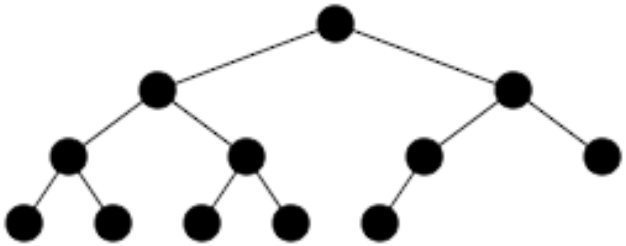
A **complete binary** tree has a restricted shape obtained by starting at the root and filling the tree by levels from left to right.

In the **complete binary tree (balanced tree)** of height **d**, all levels except possibly level **d -1** are completely full.

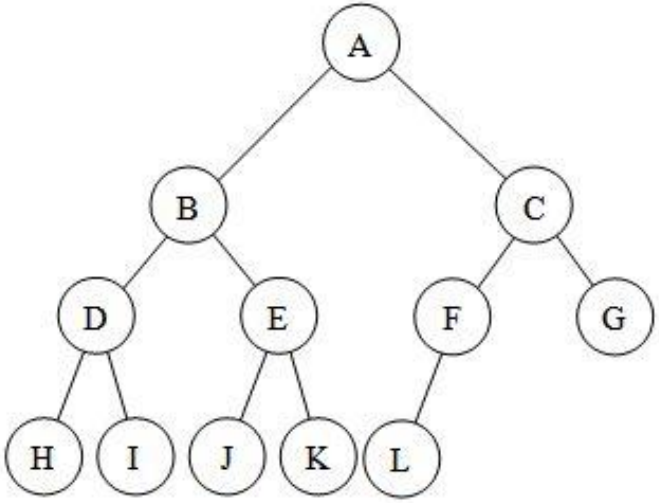
The bottom level has its nodes filled in from the left side.

Binary tree





First tree is full (but not complete). Complete tree(but not full).



Complete (balanced) trees

Binary tree implementation (1)

First algorithm

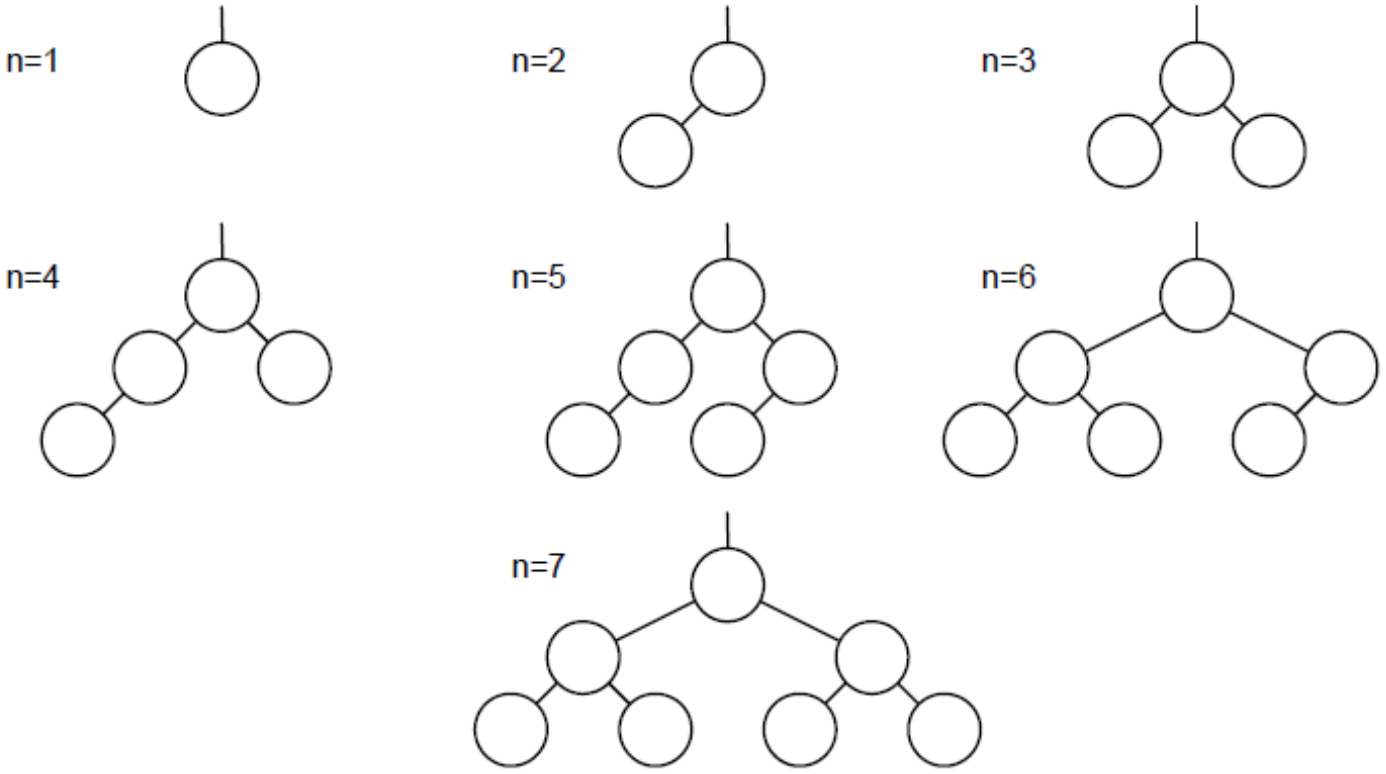
Assume we have the following data: $n_1, n_2, \ldots, n_k$.

1. n_1 – is root.
2. n_2 is compared with n_1. If $n_2 < n_1$, then n_2 is added to the left subtree, else n_2 is added to the right subtree.
3. The same principle as given in step #2 is used this the rest of the elements.

Binary tree implementation (2)

Second algorithm

Binary tree with the least height.



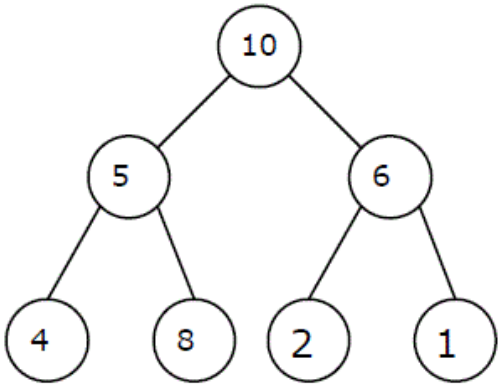
Full, complete binary tree

First, we need to work out how many nodes, ***N***, we have in such a tree of height, ***h***.

$N = 1 + 2^1 + 2^2 + ... + 2^{h-1}$

From which we have, $N = 2^h - 1$

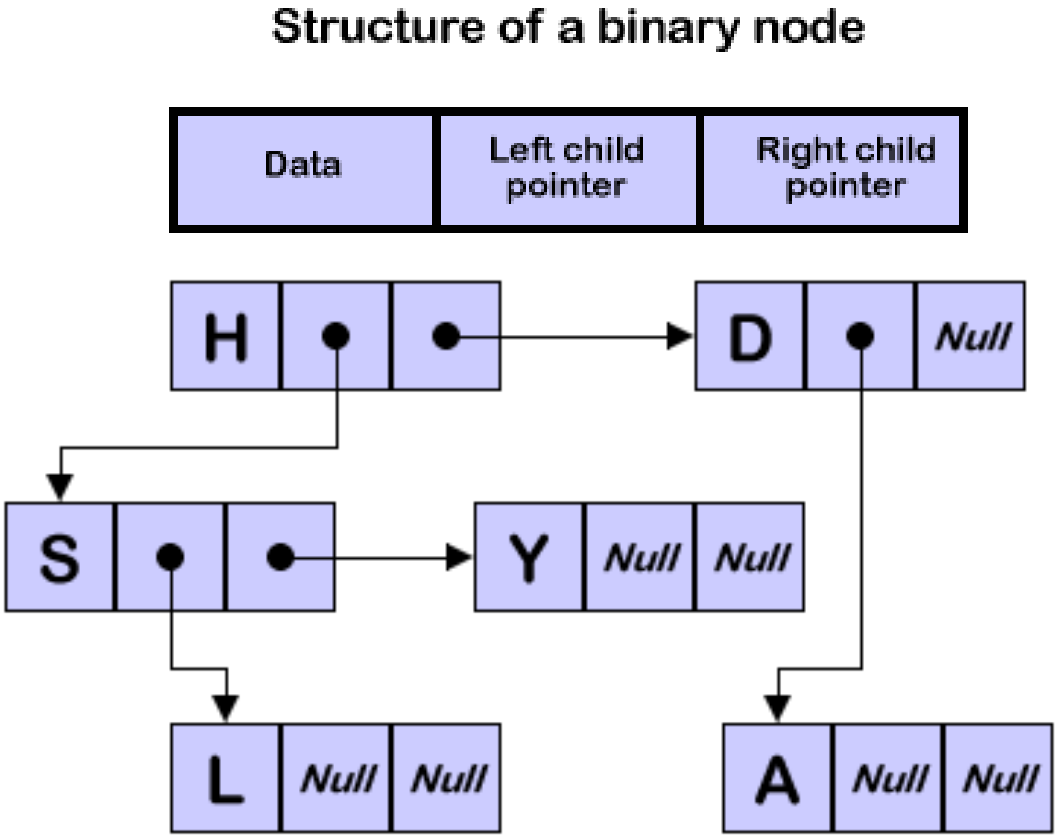
and $h = \log_2(N+1)$



Examination shows that to find any node in the tree in the worst case, takes ***h*** or **($\log_2 N$)** steps.

Structure of a binary node

Node of binary tree needs to reserve memory for the data and two pointers (for pointing two childs of that node).



Space Requirements

In a simple pointer-based implementation for the binary tree, every node has two pointers to its children (even when the children are **NULL**).
This implementation requires total space amounting to :

N*(2P + D)

here N – number of nodes, P stands for the amount of space required by a pointer, and D stands for the amount of space required by a data value.

The total overhead space will be **2*P*N** for the entire tree.

Space Requirements

Thus, the overhead fraction will be: 2P/(2P + D).

The actual value for this expression depends on the relative size of pointers versus data fields. If we arbitrarily assume that P = D, then a full tree has 2/3 of its total space taken up in overhead.

Great savings can be had by eliminating the pointers from leaf nodes in **full binary trees**. Because about half of the nodes are leaves and half internal nodes, and because only internal nodes now have overhead, the overhead fraction in this case will be approximately:

$$\frac{\frac{n}{2}(2P)}{\frac{n}{2}(2P) + Dn} = \frac{P}{P + D}.$$

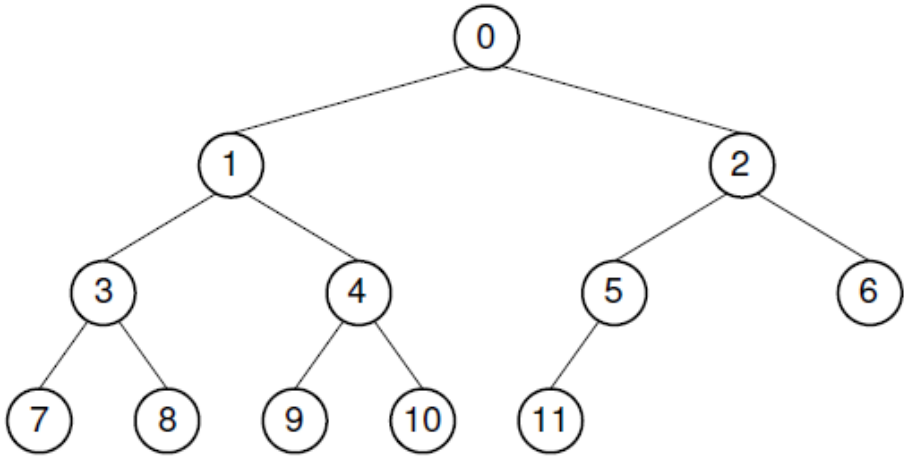
If P = D, the overhead drops to about one half of the total space

Implentation of complete tree

```
node* balancedTree (int N)      // N- number of nodes
{
   if ( N == 0 )
      return (NULL);
  nL = N/2; nR = N - nL -1;
  x = read();
  node* Node= (*node) malloc(sizeof(node));
  (*Node).data = x;
  (*Node).left = balancedTree(nL);
  (*Node).right = balancedTree(nR);
  return(Node);
}
```

Array Implentation for trees

Compact implementation for complete binary trees is based on arrays.



(a)

Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

Array implementation of the tree

The formulae for calculating the array indices of the various relatives of a node are as follows. The total number of nodes in the tree is *n*. The index of the node in question is *r*, which must fall in the range 0 to n - 1.

- $\text{Parent}(r) = \lfloor (r-1)/2 \rfloor$ if $r \neq 0$.
- $\text{Left child}(r) = 2r + 1$ if $2r + 1 < n$.
- $\text{Right child}(r) = 2r + 2$ if $2r + 2 < n$.
- $\text{Left sibling}(r) = r - 1$ if r is even.
- $\text{Right sibling}(r) = r + 1$ if r is odd and $r + 1 < n$.

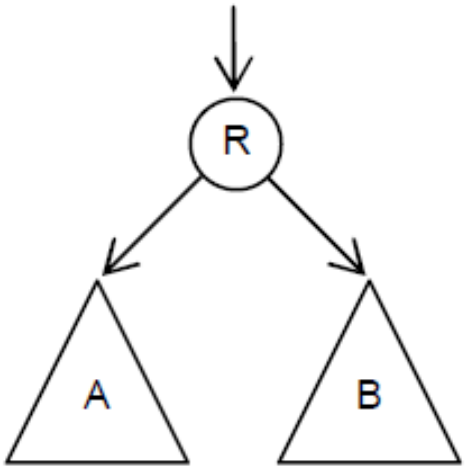
Binary Tree Traversals

Often we wish to process a binary tree by “visiting” each of its nodes, each time performing a specific action such as printing the contents of the node.

Any process for visiting all of the nodes in some order is called a **traversal**.

Any traversal that lists every node in the tree exactly once is called an **enumeration of the tree’s nodes**.

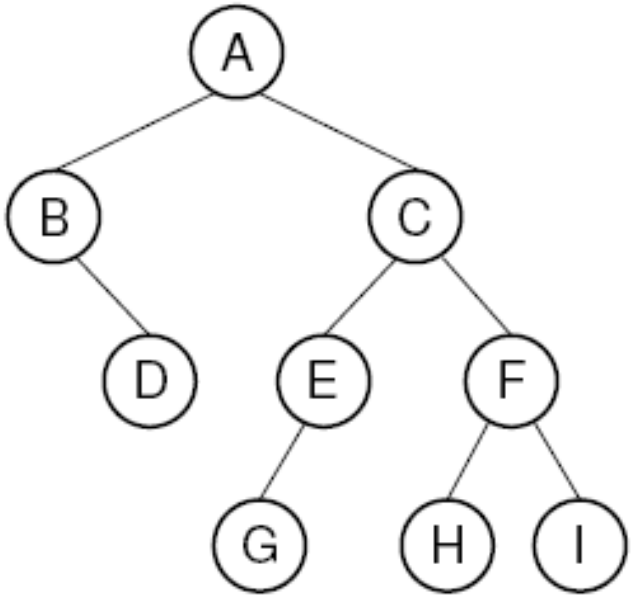
Enumeration



There are three principal enumerations that emerge naturally from the structure of trees.

1. **Pre**order: R, A, B (visit root before the subtrees)
2. **In**order: A, R, B
3. **Post**order: A, B, R (visit root after the subtrees)

Example

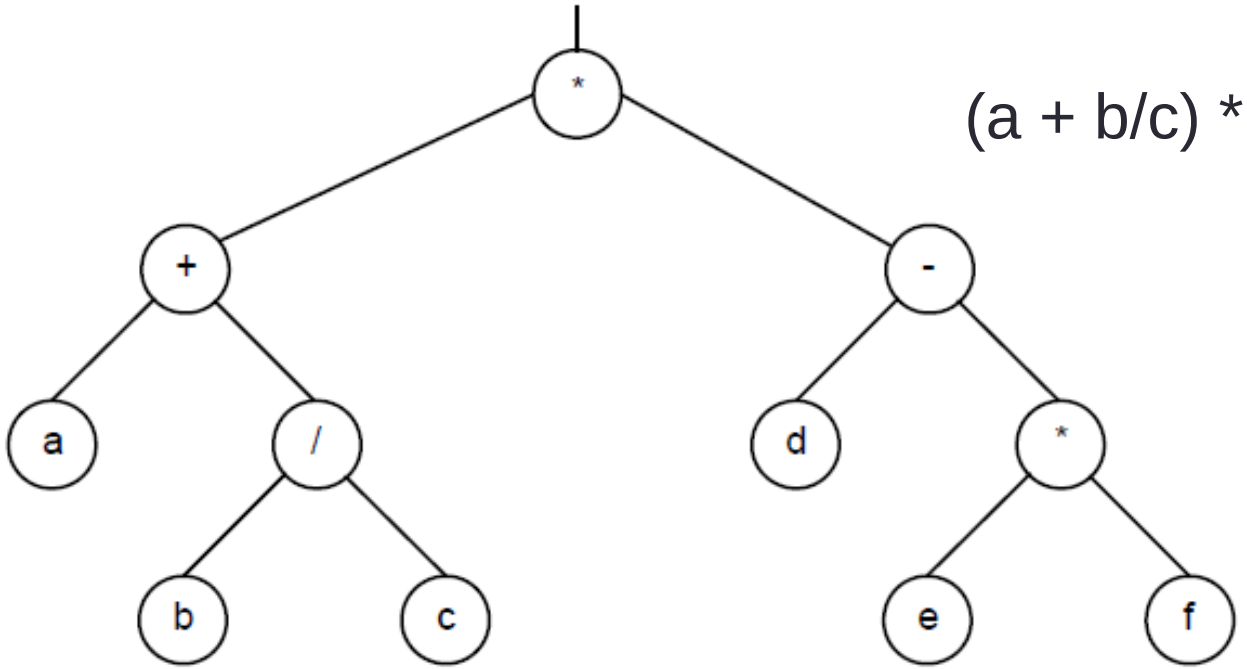


Preorder enumeration: ABDCEGFHI

Inorder enumeration : DBAGECHFI

Postorder enumeration: DBGEHIFCA

Example



(a + b/c) * (d – e*f)

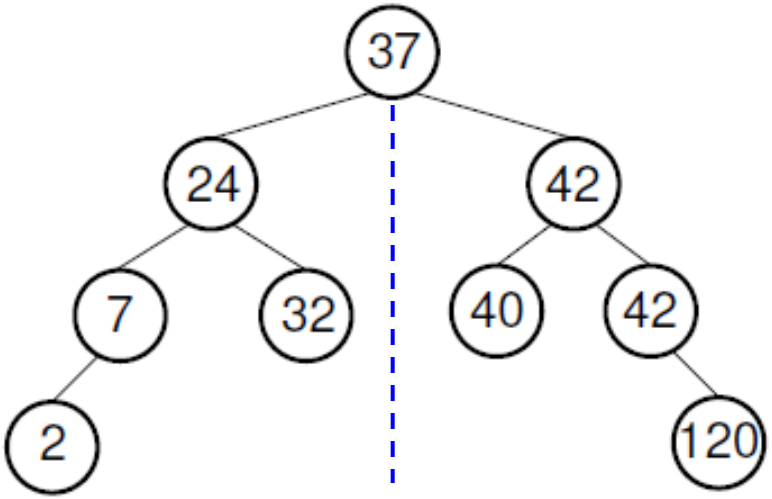
1. Preorder: *** + a / b c - d * e f**
2. Inorder: **a + b / c * d - e * f**
3. Postorder: **a b c / + d e f * - ***

Binary search tree (BST)

A BST is a binary tree that conforms to the following condition, known as the Binary Search Tree Property:

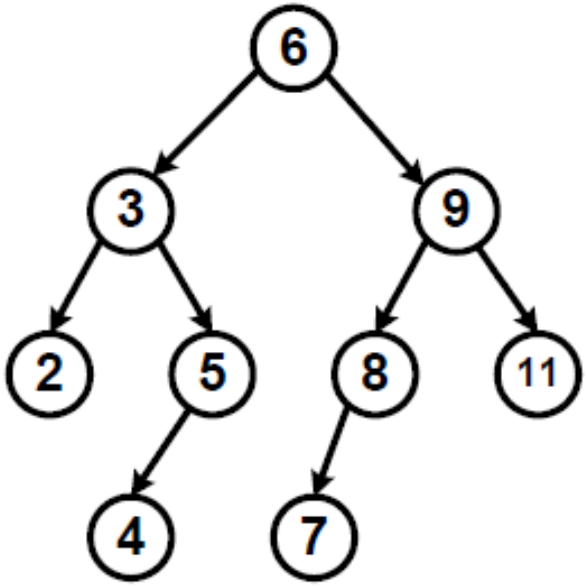
- All nodes stored in the left subtree of a node whose key value is K have key values less than K.
- All nodes stored in the right subtree of a node whose key value is K have key values greater than or equal to K.

All nodes less then root

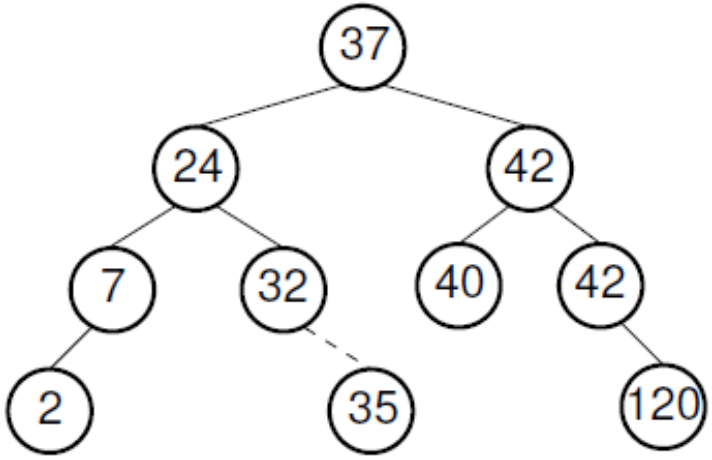


All nodes greater then root

Binary search tree



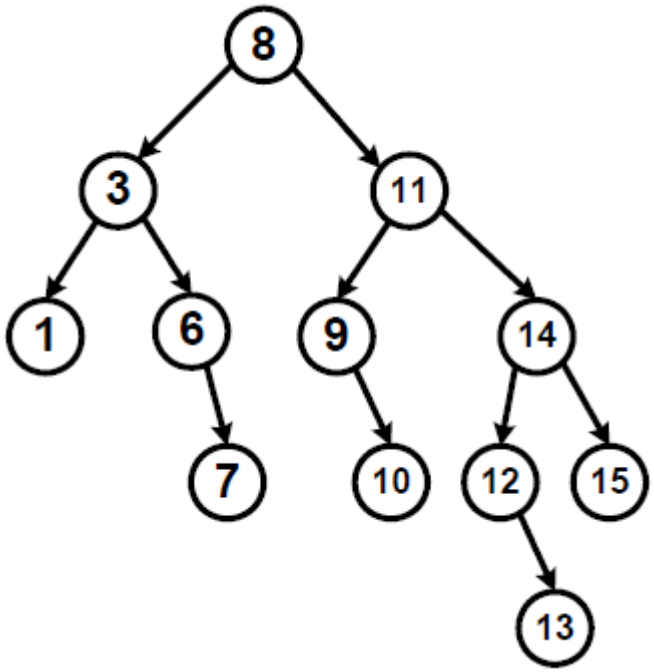
Binary search tree



Node insertion

Building binary search tree

Initial data: 8, 11, 9, 3, 1, 14, 6, 12, 10, 7, 13, 15



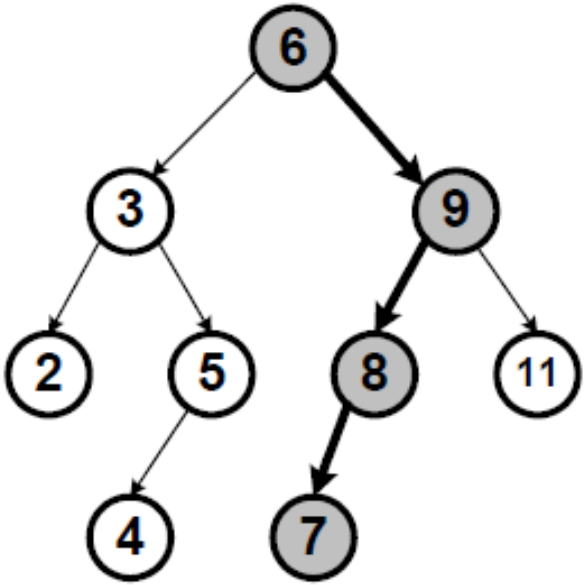
Principle used to build BST:

Inequality $p_i < p_k$ defines the which subtree to choose at each level.

Comparison of the numbers is starting from the root node.

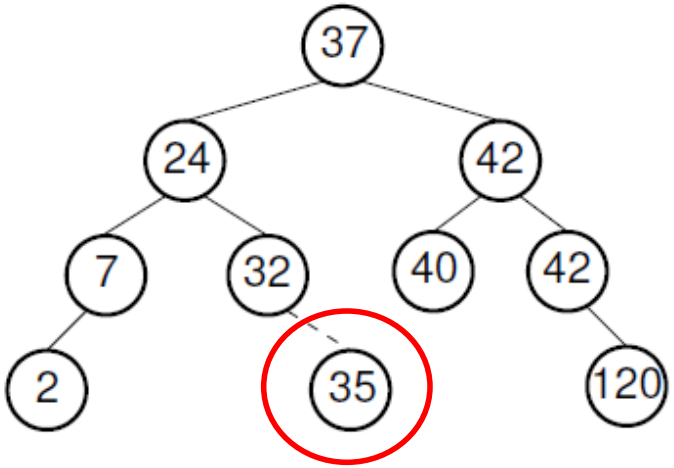
Node search

```
node* find_node (node* tree, int find_data)
{
     while ( (tree != NULL) && ((*tree).data != find_data) )
    {
            if ( (*tree).data < find_data )
                  tree = (*tree).right;
            else
                  tree = (*tree).left;
     }
   return tree;
}
```

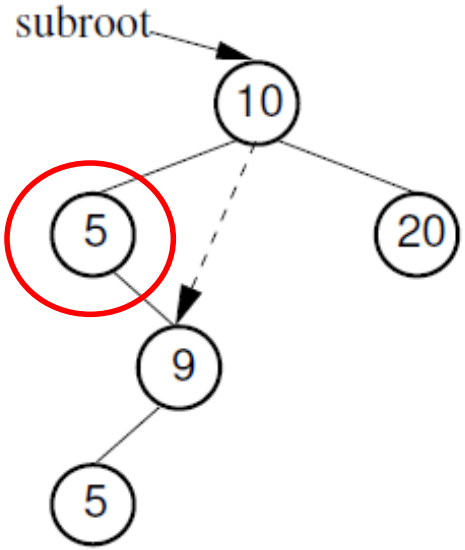


Insert new node

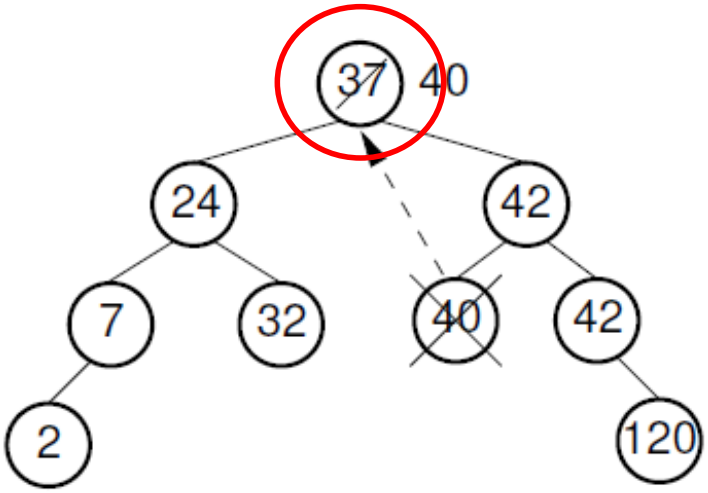
```
insert_node (node* tree, node* v)
{
        while ( tree != NULL )
        {
                if ( (*v).data < (*tree).data )
                        tree = (*tree).left;
                else
                        tree = (*tree).right;
        }
  tree = v;
  (*v).left = NULL;
  (*v).right = NULL;
}
```



Deleting BST node



Deleting the node with minimum value. In this tree, the node with minimum value, 5, is the left child of the root. Thus, the root's **left** pointer is changed to point to 5's right child.



Removing the value 37 from the BST. The node containing this value has two children. We replace value 37 with the least value from the node's right subtree, in this case 40.

Delete BST node

```
deleteNode(node* v, int a)  {
node* p, q;
v = find(v, a);
if ( v != NULL ) {                // Node has found
    q = v;
if ( (*v).left == NULL )  {    // Not more then one child
    v = (*v).right;
}  }
else
    if ( (*v).right == NULL ) {  // Just left child
       v = (*v).left;
}
 else {                              // if v has two children, next node must be found
     p = (*v).right;
       while ( (*p).left != NULL )
             p = (*p).left;
             v = p;
             p = (*p).right;    (*v).left = (*q).left;
            (*v).right = (*q).right;   }   delete q; }
```

Homework

No 1. Write a C program that implement the following funcionality:

- Generate 10 random numbers and find numbers that are not factorial;
- Use these numbers (not factorials) to build binary tree (array implementation).

No 2. Write a C program to merge two singly linked lists. Linked lists can have different number of nodes.