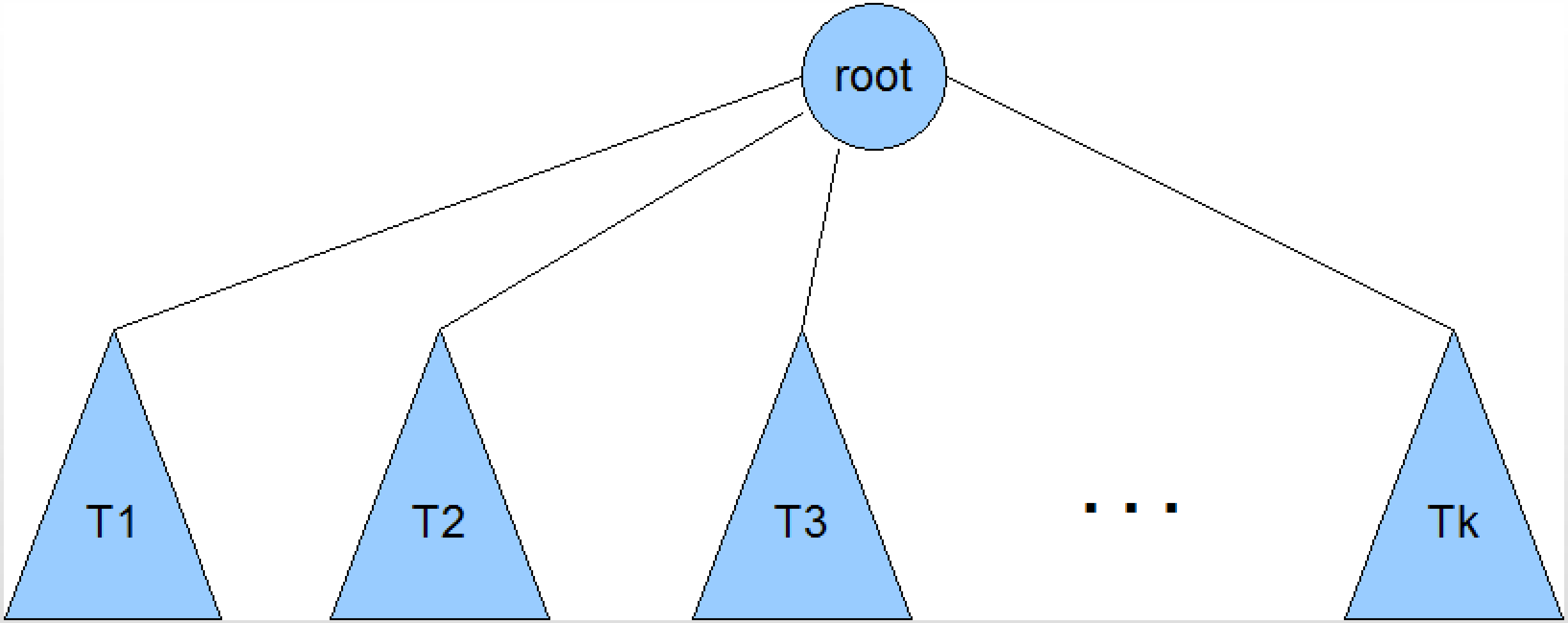
4. Trees

4.1 Basic Concepts and Terminology

Trees

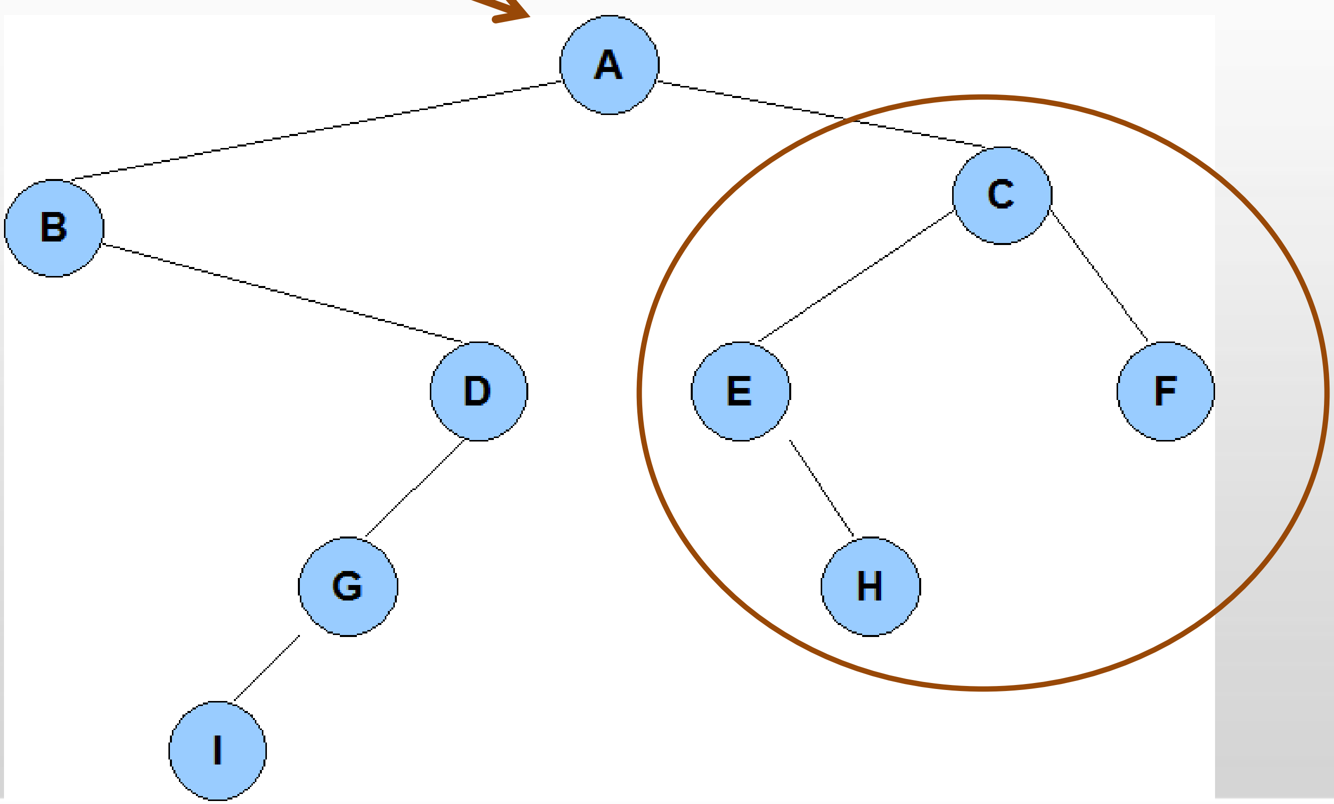
- A tree is a collection of nodes.
- The collection can be empty, otherwise, a tree consists of a distinguished node r, called the root, and zero or more (sub)trees T_1, T_2, …, T_k, each of whose roots are connected by a directed edge to r.
- no. of nodes = N, no. of edges = ?

Generic Tree



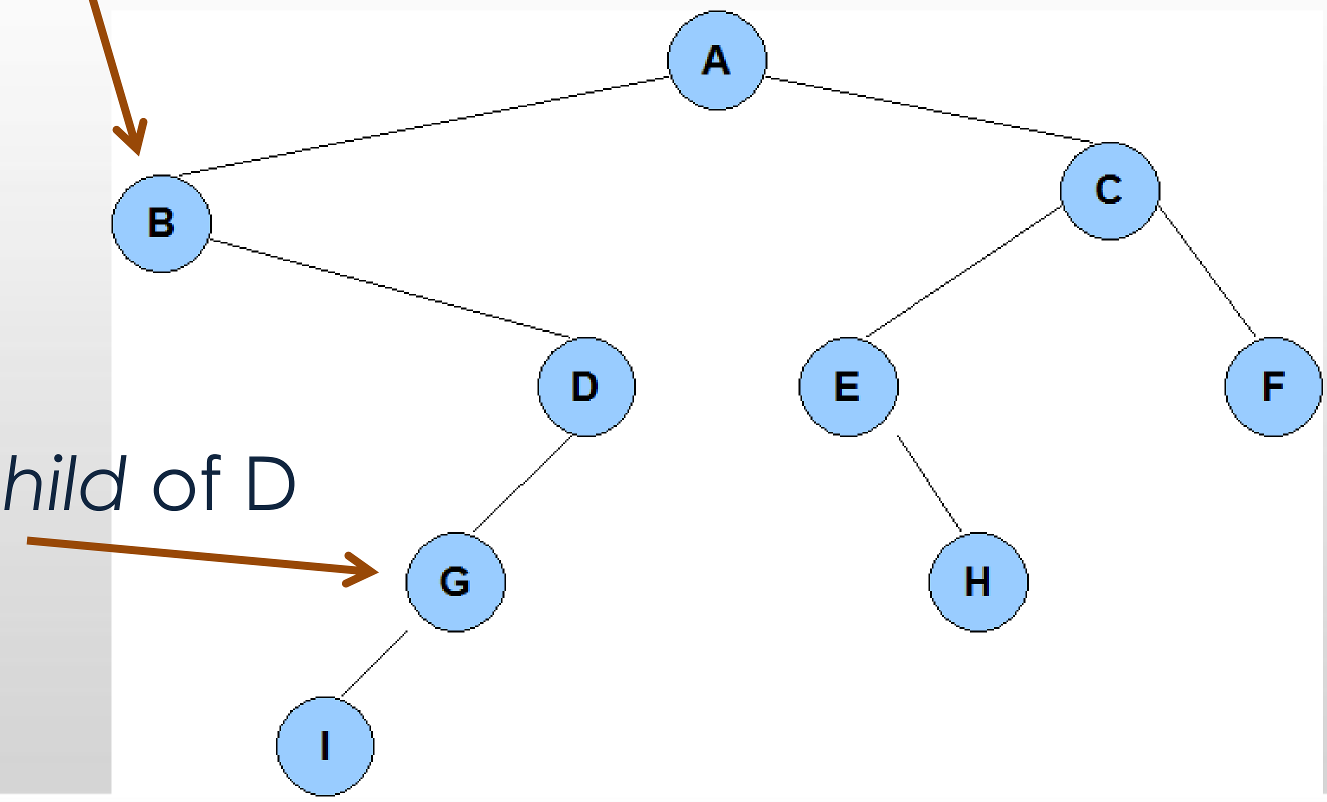
Trees

- The *root* of each subtree is said to be a *child* of r, and r is the *parent* of each subtree root.



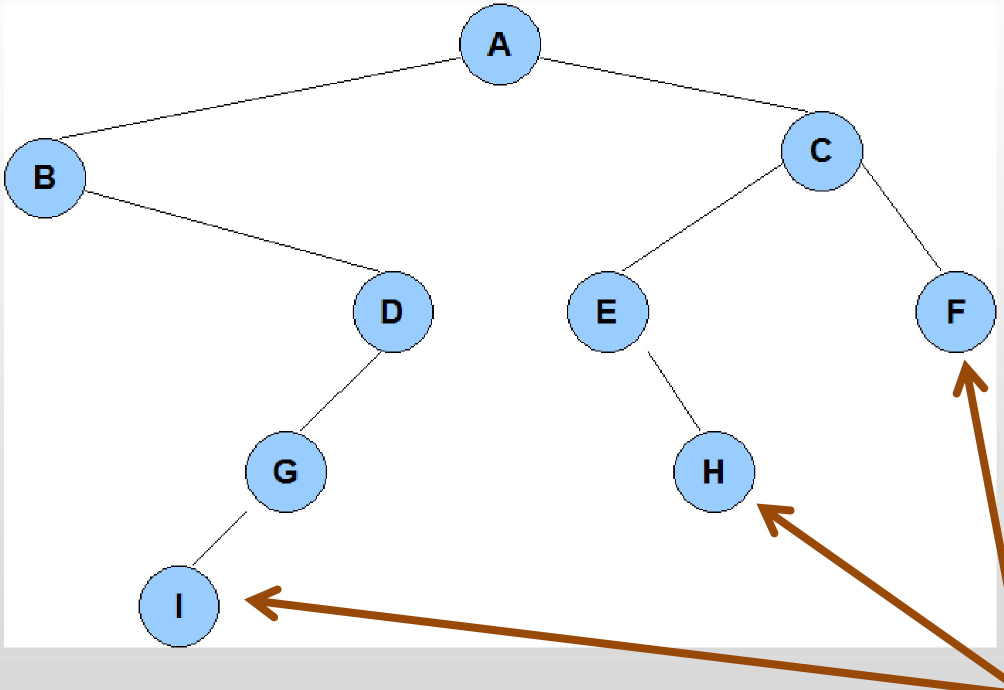
Trees

- *Parent* of D
- *Child* of D



Trees

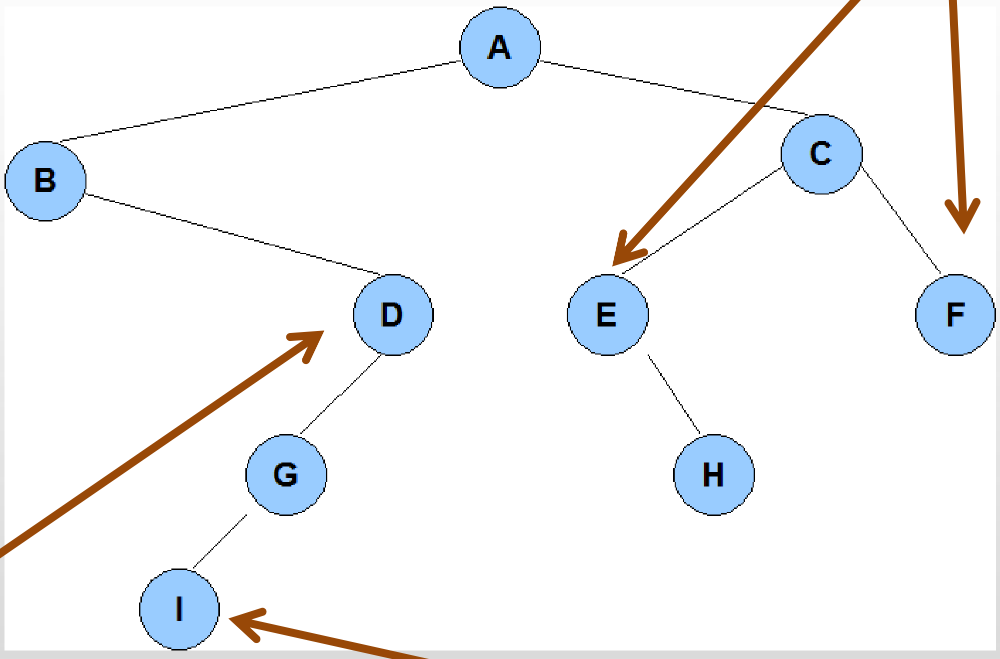
- Each node may have an arbitrary number of children, possibly zero.



- Nodes with no children are known as *leaves*.

Trees

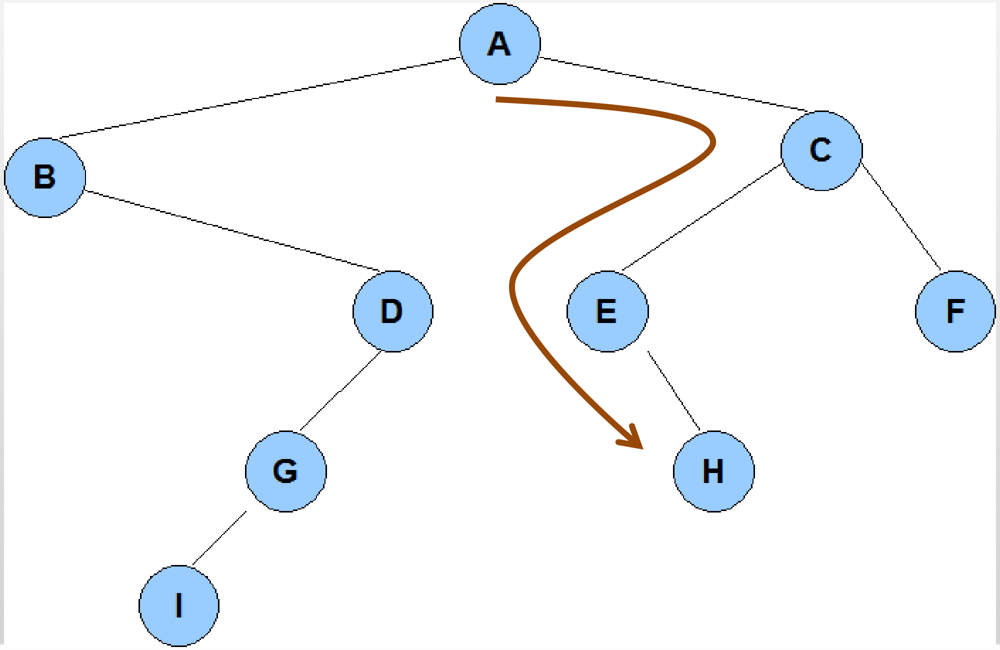
- Nodes with the same parent are *siblings*.



- *Grandparent* and *grandchild*

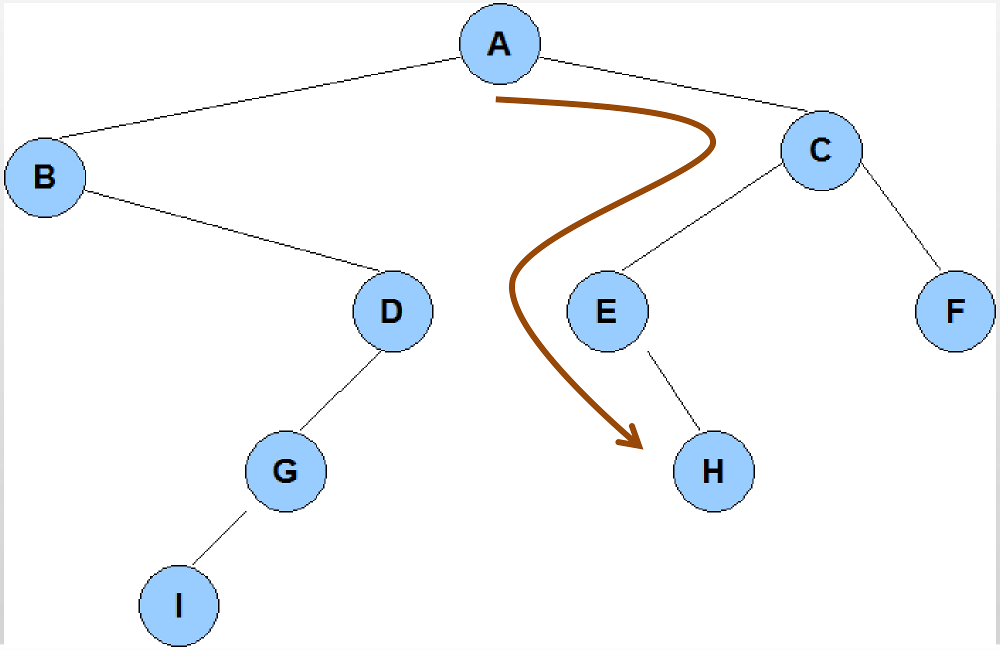
Trees

- A *path* from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 <= i <= k$.



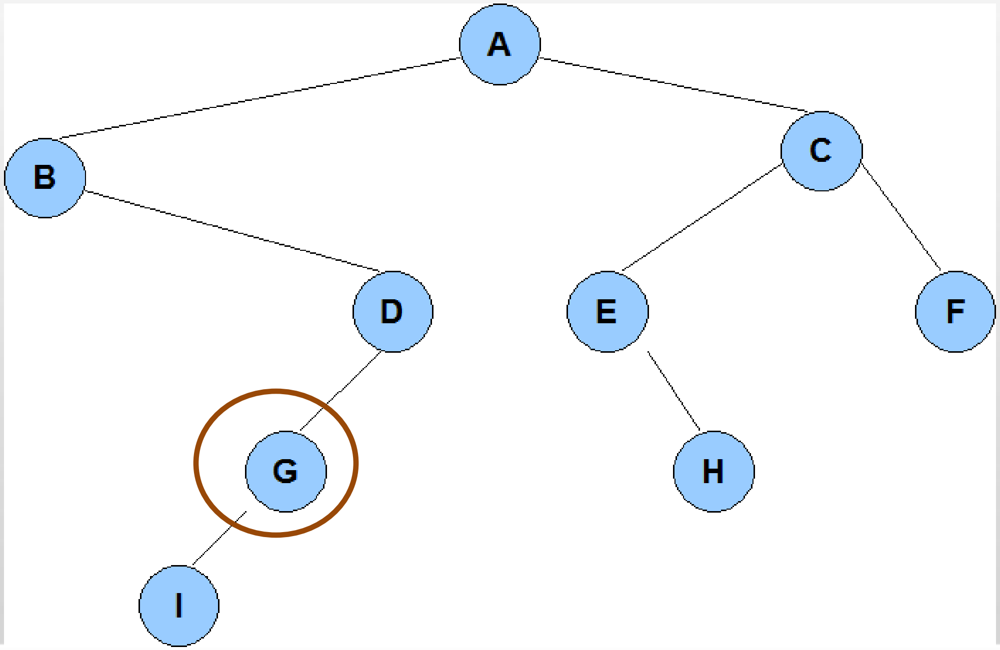
Trees

- The *length* of a path is the number of edges on the path.
- There is a path of length zero from every node to itself.



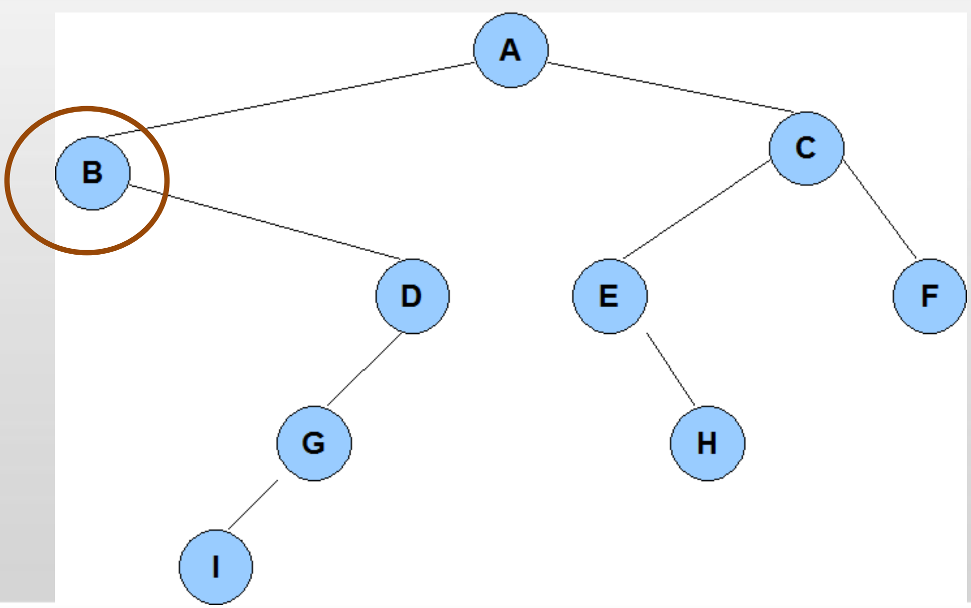
Trees

- For any node n_i, the *depth* of n_i is the length of the unique path from the root to n_i.
- Thus the root is at depth 0.



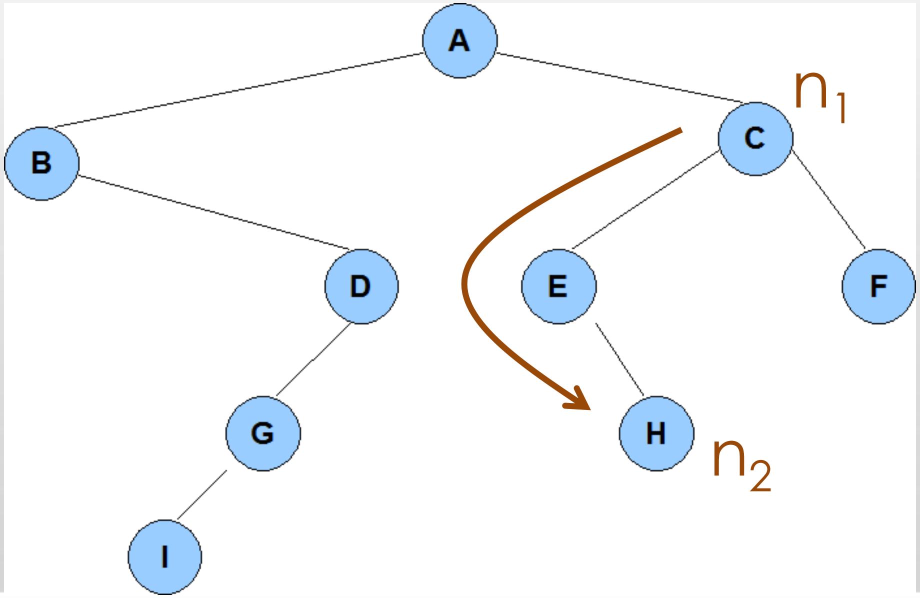
Trees

- The *height* of n_i is the longest path from n_i to a leaf.
- Thus all leaves are at height 0.
- The height of a tree is equal to the height of the root.

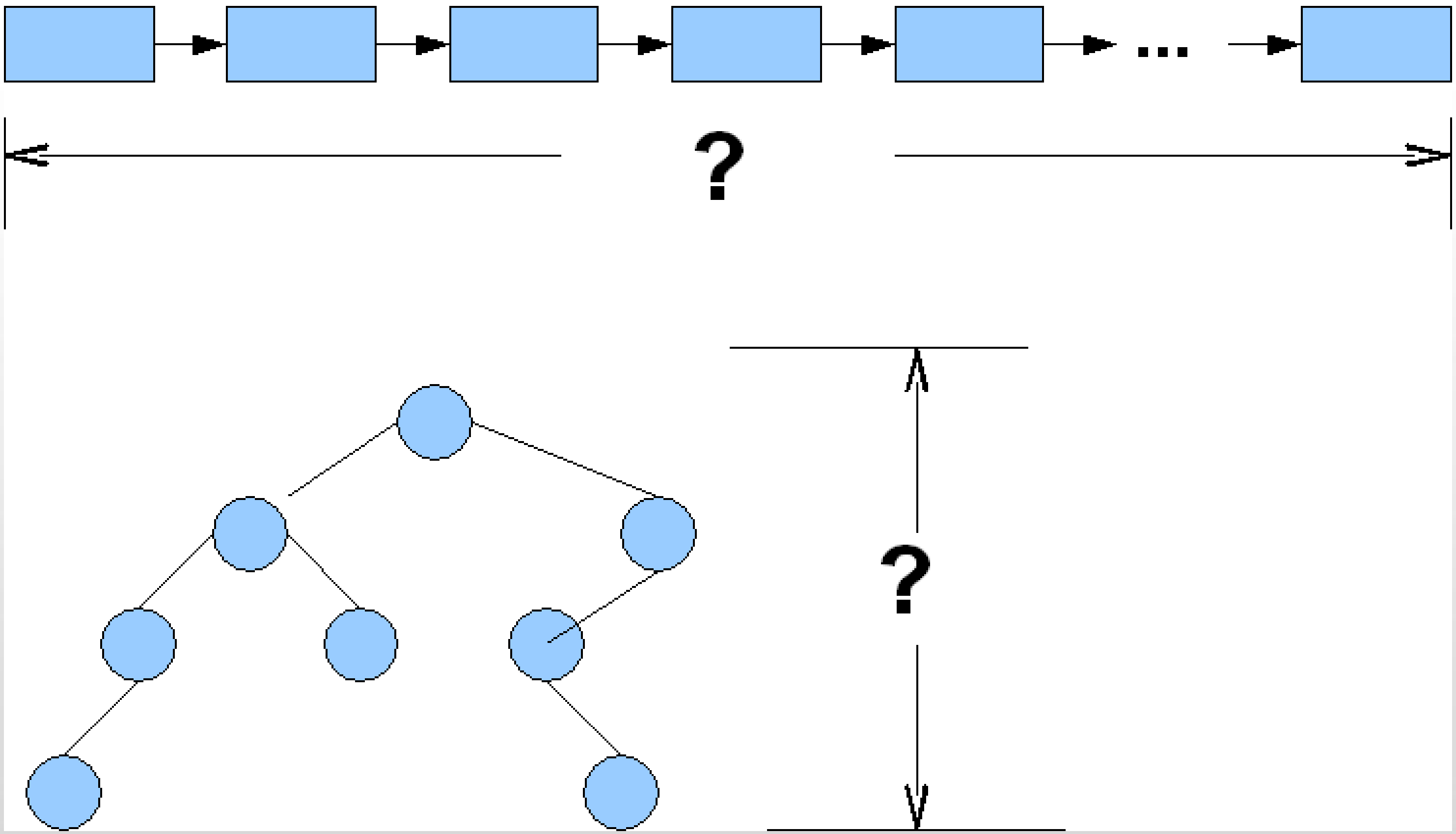


Trees

- If there is a path from n_1 to n_2, then n_1 is an *ancestor* of n_2 and n_2 is a *descendant* of n_1.
- If $n_1 \neq n_2$, then n_1 is a *proper ancestor* of n_2 and n_2 is a *proper descendant* of n_1.



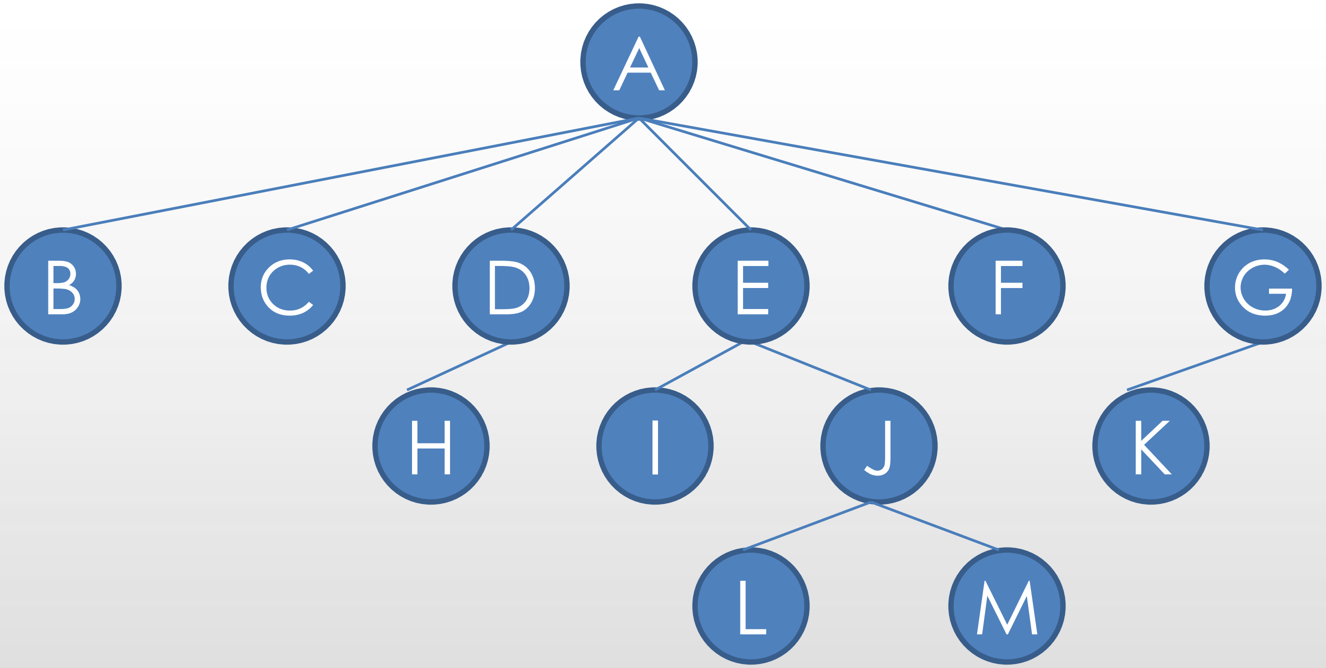
Tree vs. Linear List



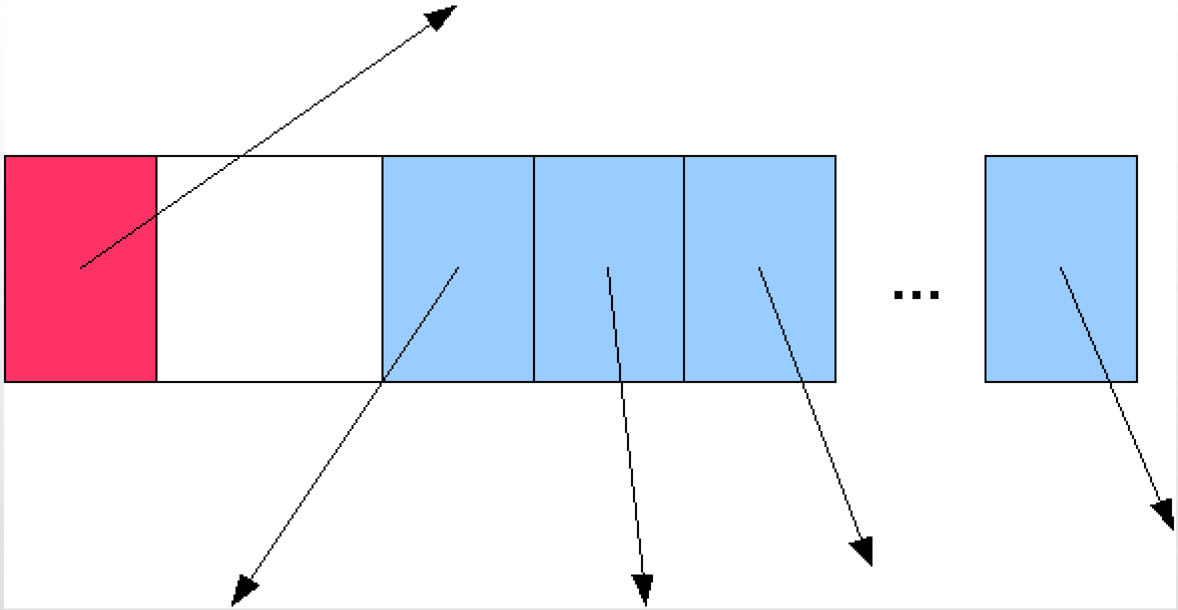
Implementation of Trees

- Linked representation
  - each node, besides its data has a pointer to each child of the node
- Left-child, Right-sibling representation
  - children of each node is kept in a linked list of tree nodes

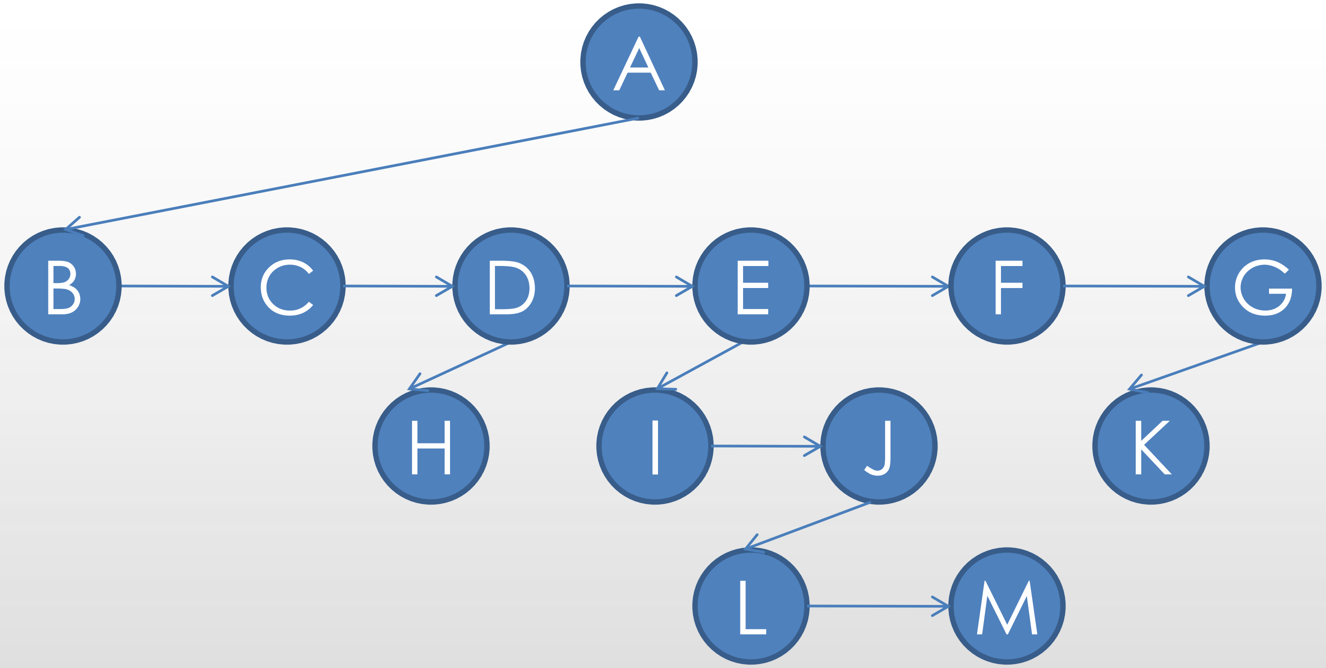
Linked Representation



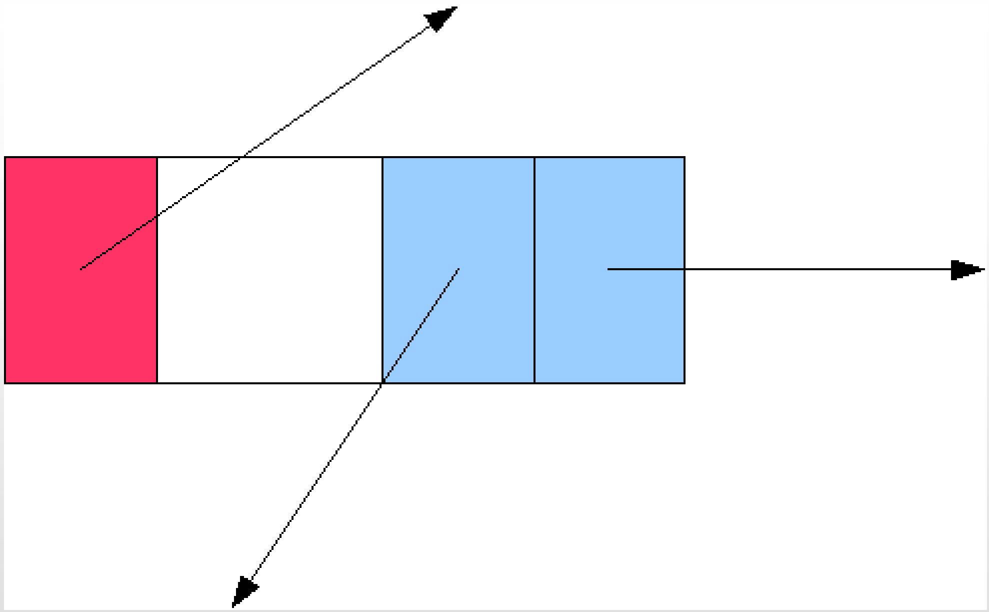
Linked Representation



Left-child Right-sibling Representation



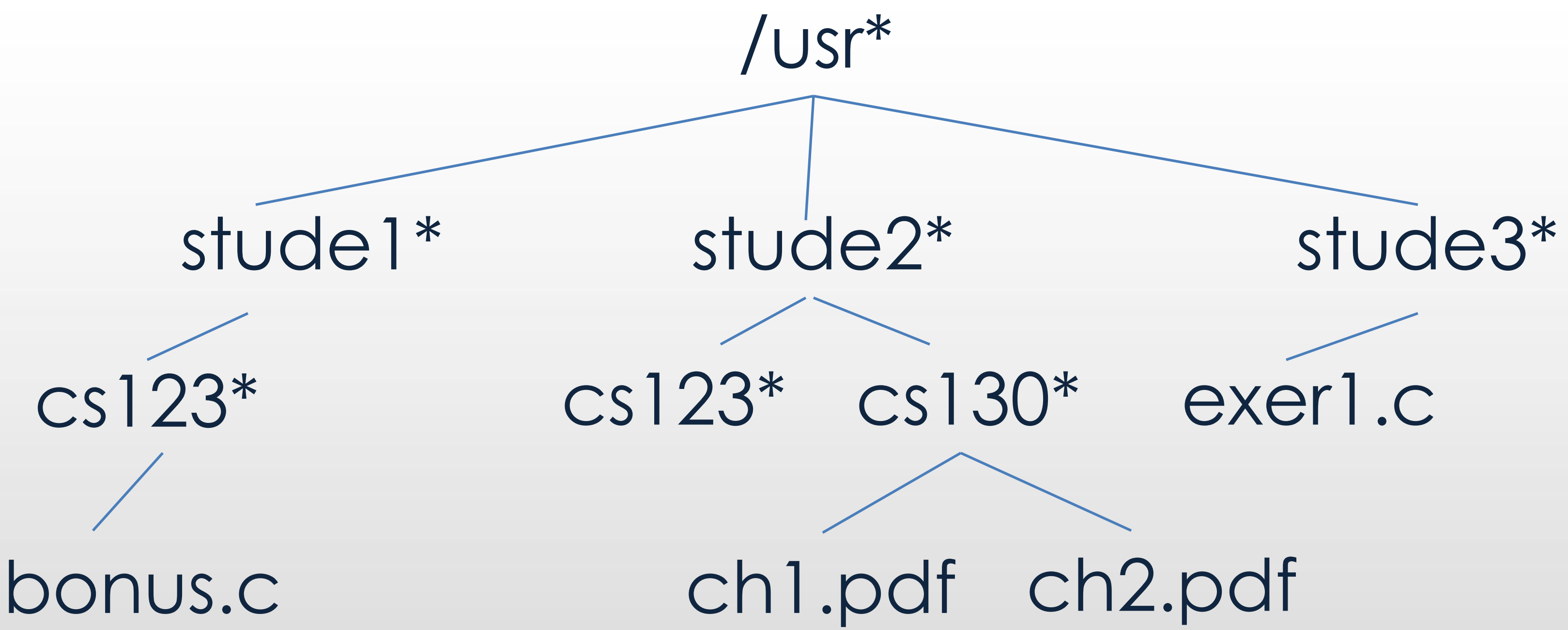
Left-child Right-sibling Representation



Applications

- Directory listing
- Getting the directory/file sizes
- Compiler design (e.g. Expression trees)

Directory Listing



- /usr/stude1/cs123/bonus.c

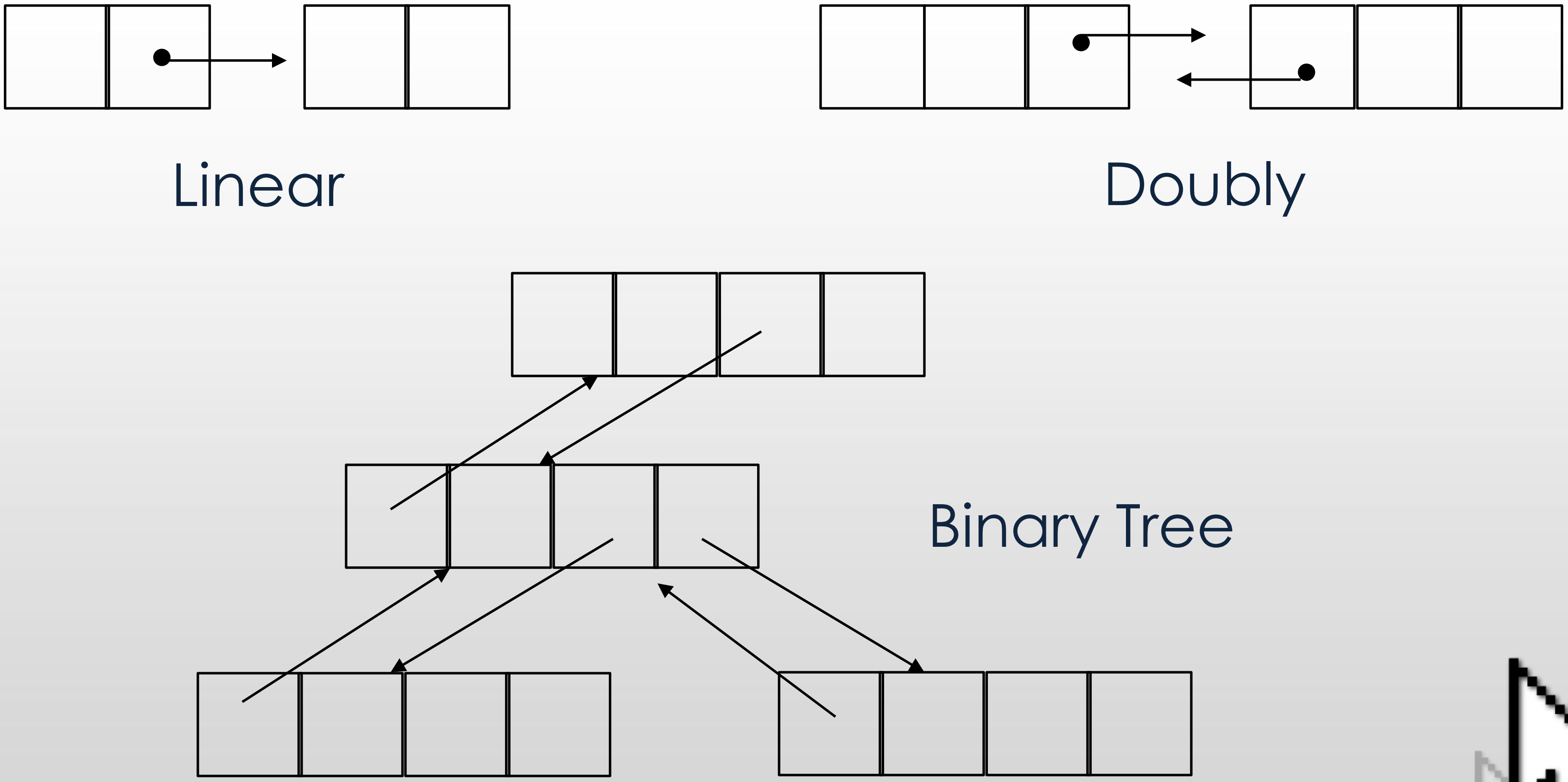
4. Trees

4.2 Binary trees and their implementations

Binary Tree

- Linear linked list – node has at most one pointer to another node
- Doubly linked list – node has at most two pointers to two different nodes
- Binary tree – node has at most three pointers to three different nodes with each node having a back pointer

Binary Tree

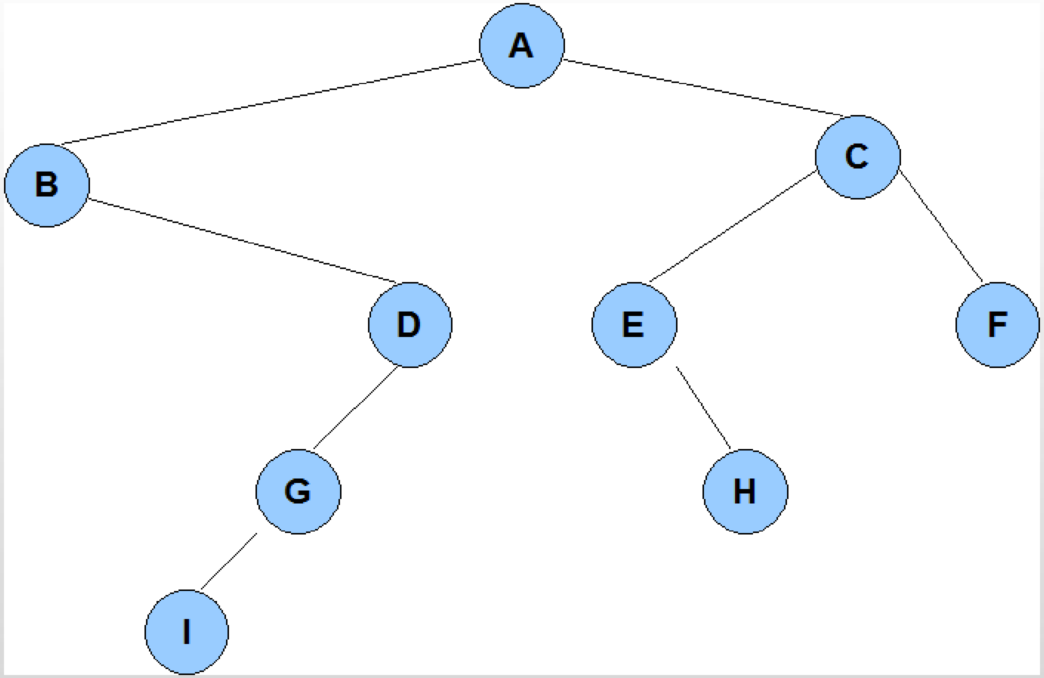


Binary Tree

- A structure that is either empty
- Or one that contains a single node called the root
- Or a root with at most two descendants, each descendant being a root of another binary tree

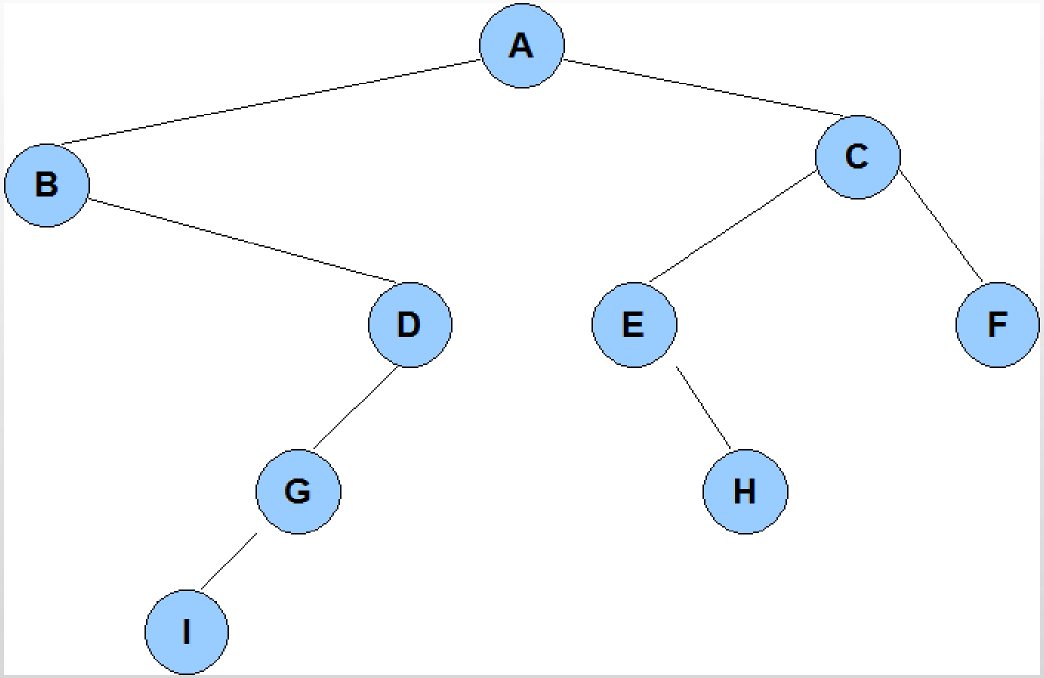
Binary Tree

- *Internal nodes* – nodes with at least one child



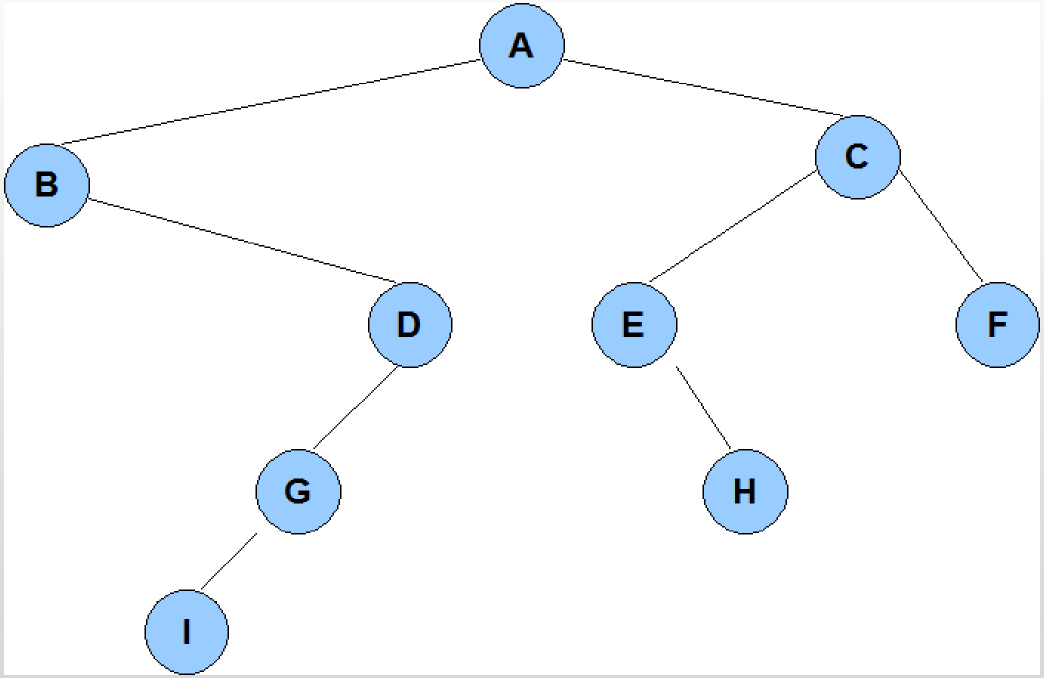
Binary Tree

- *Level* of a node – 1 plus the level of its parent (root has level 0)



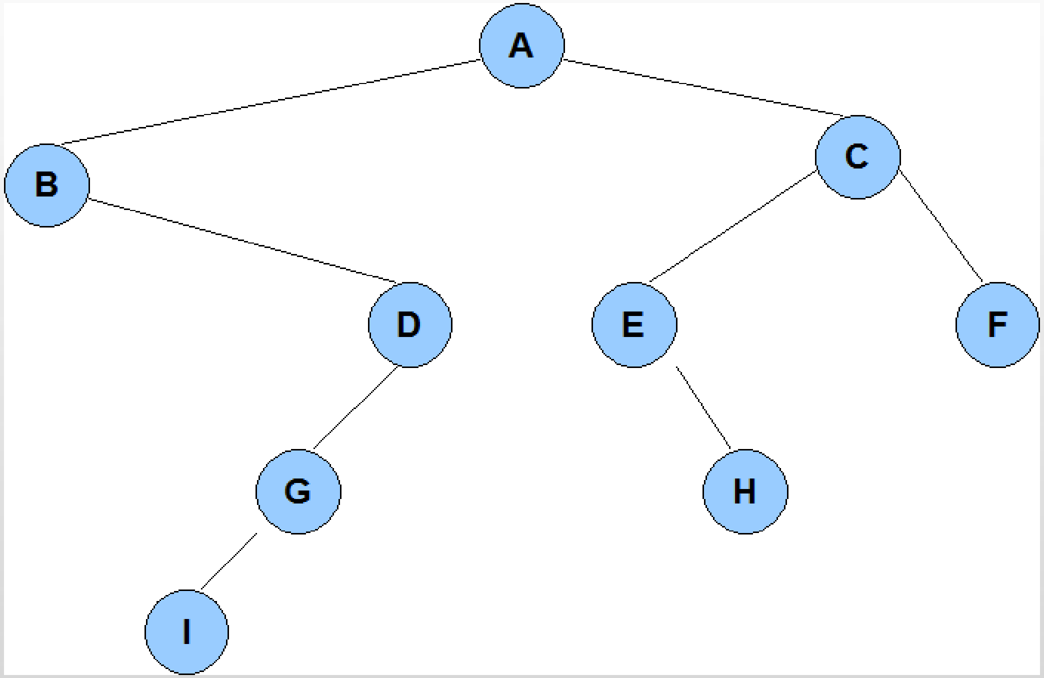
Binary Tree

- *Level i is full* if there are exactly 2^i nodes at this level



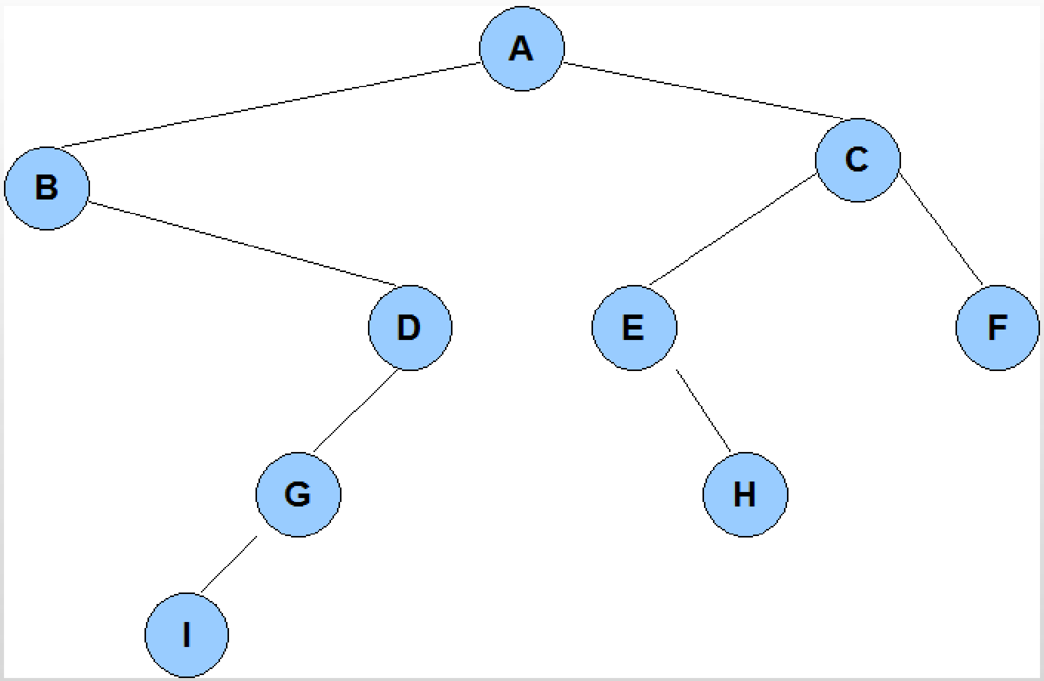
Binary Tree

- A *binary tree of height k is full* if level k in this binary tree is full



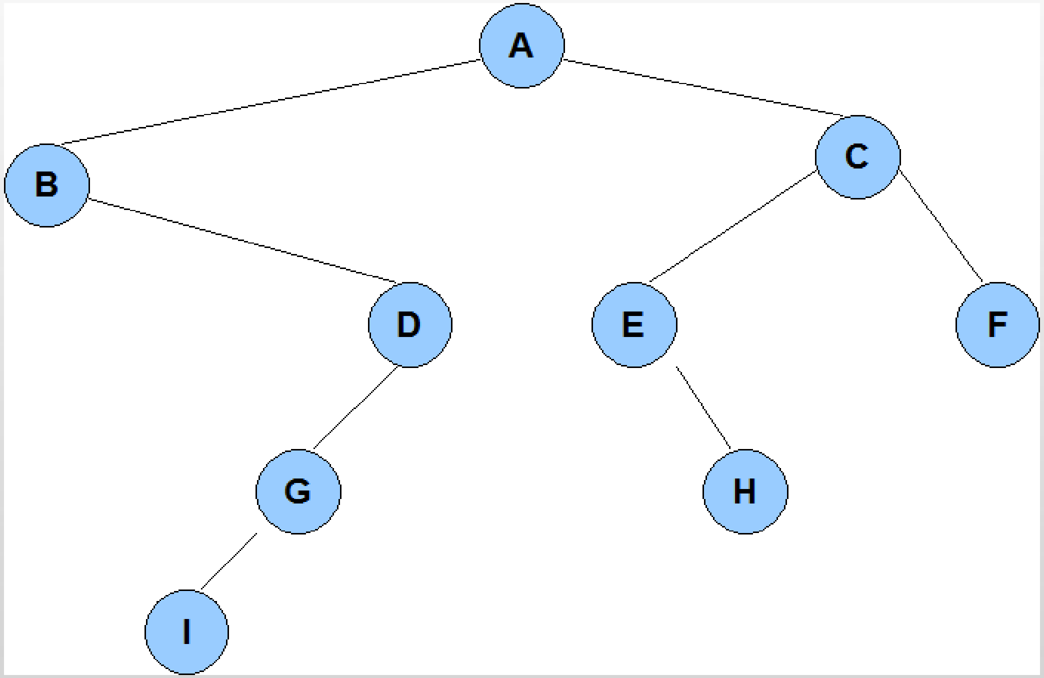
Binary Tree

- A binary tree of height k is *balanced* if level k-1 in this binary tree is full



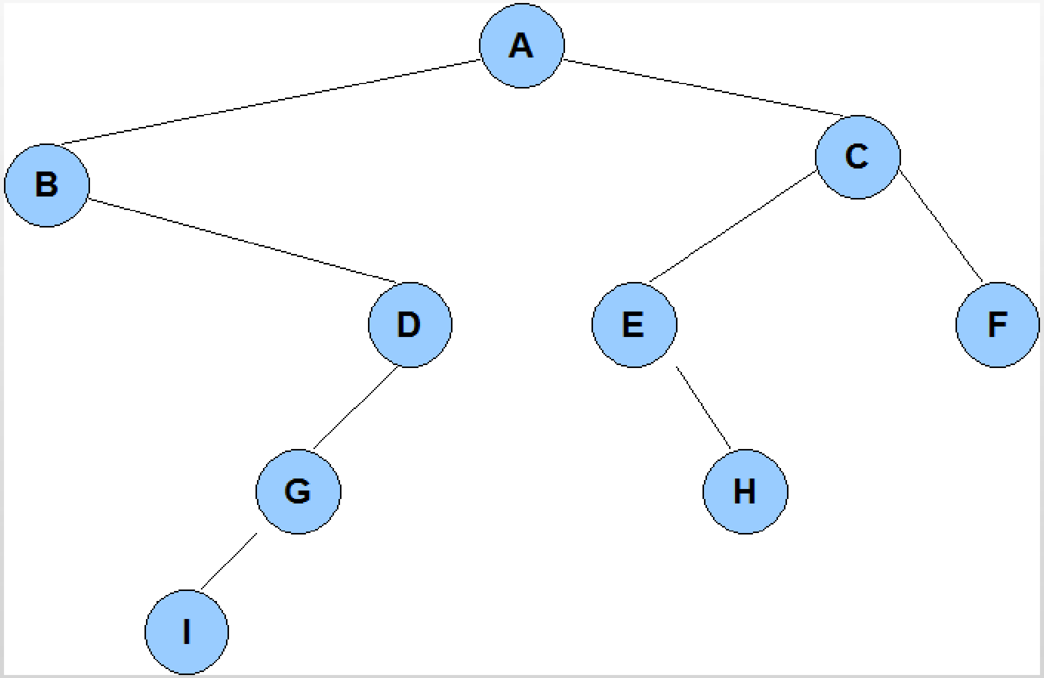
Binary Tree

- The binary tree rooted at the left child of node v is the *left subtree* of v, and that rooted at the right child is the *right subtree*



Binary Tree

- A binary tree is *height balanced* if, at every node in the tree, its left and right subtrees differ by no more than 1 in height



Binary Tree

- Each node contains at least the value field, left child pointer, right child pointer, and a pointer to its parent
- If a parent or a child is not present, then the value of the pointer is NULL

Binary Tree

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}tree;
```

Binary Tree Traversal

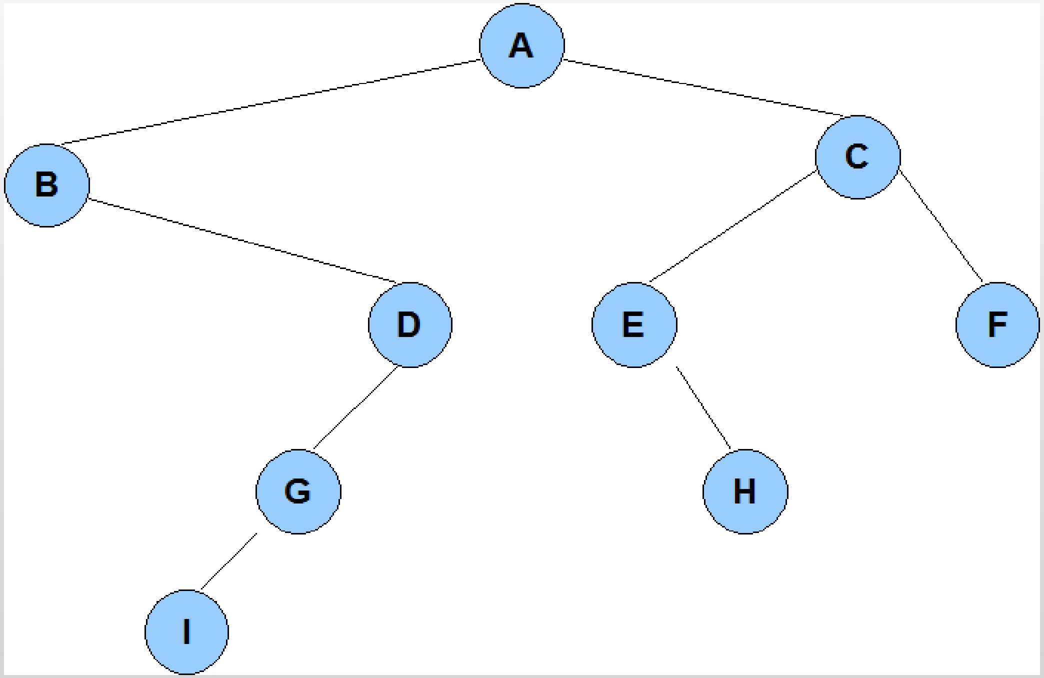
- Method of visiting each node of the tree at least once
- 3 ways:
  - Inorder
  - Preorder
  - Postorder

Binary Tree Traversal

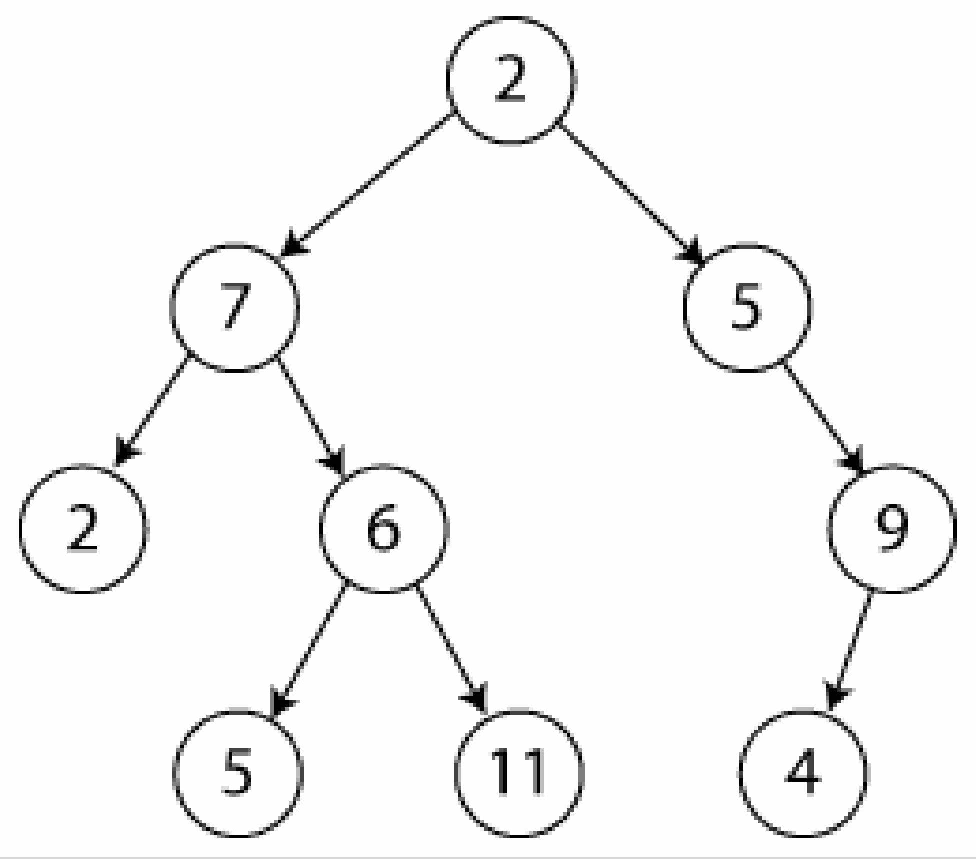
- Inorder
  - Left subtree is first visited, then the root node, and finally the right subtree
- Preorder
  - Root node is visited first, then left subtree, and finally the right subtree
- Postorder
  - Left subtree is visited first, then right subtree, and finally the root

Binary Tree

- Inorder: B I G D A E H C F
- PreOrder: A B D G I C E H F
- PostOrder: I G D B H E F C A

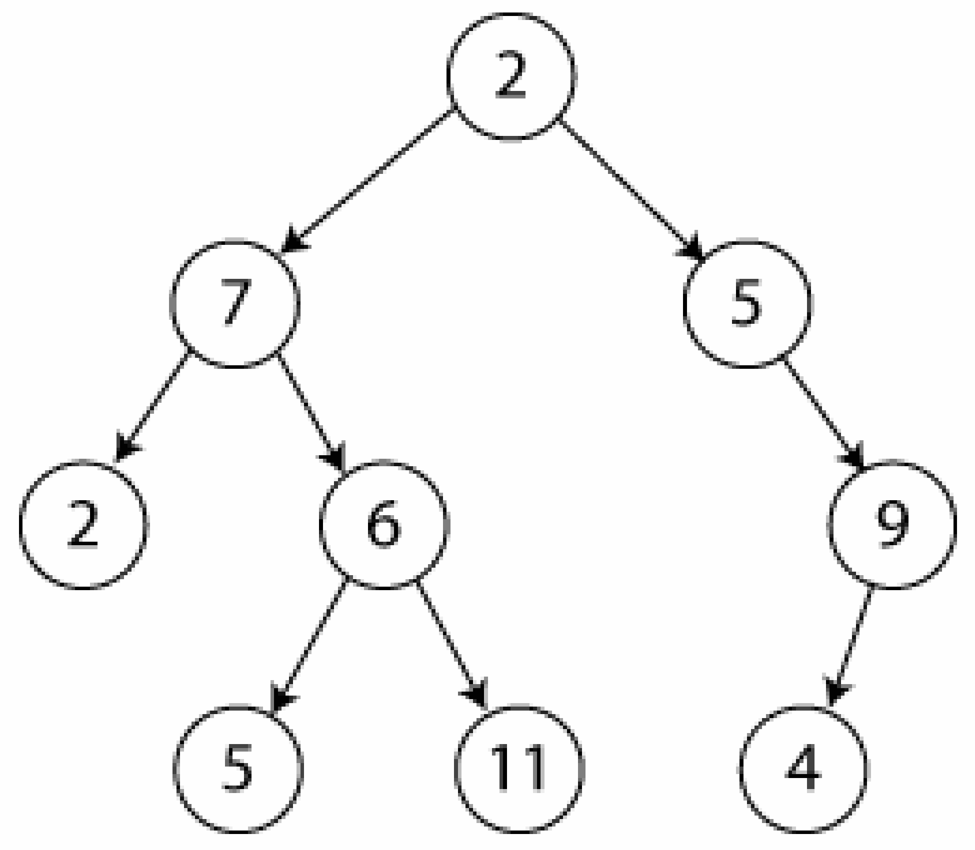


Recall



- root = 2
- parent of 6 = 7
- child of 9 = 4
- leaves = 2, 5, 11, 4
- sibling of 7 = 5
- path from 7 to 11
  = 7, 6, 11
- length of path = 2

Recall



- depth of 4 = 3
- height of 7 = 2
- height of tree = 3
- level of 11 = 3
- Is level 2 full? No
- Is the tree full? No
- Is it balanced? No
- Is it height balanced? No

Expression Trees

- A binary tree where the leaves hold the operands of the expression and the internal nodes hold the operators of the expression.
- The algorithm for creating an expression tree from the postfix form of the expression will require a stack of pointers to binary trees.

Expression Trees

Algorithm

1. Read a symbol from the postfix form of the expression.
2. If the symbol is an operand, create a binary tree with one node from the operand. Then push into the stack a pointer to this one-node tree.

Expression Trees

Algorithm

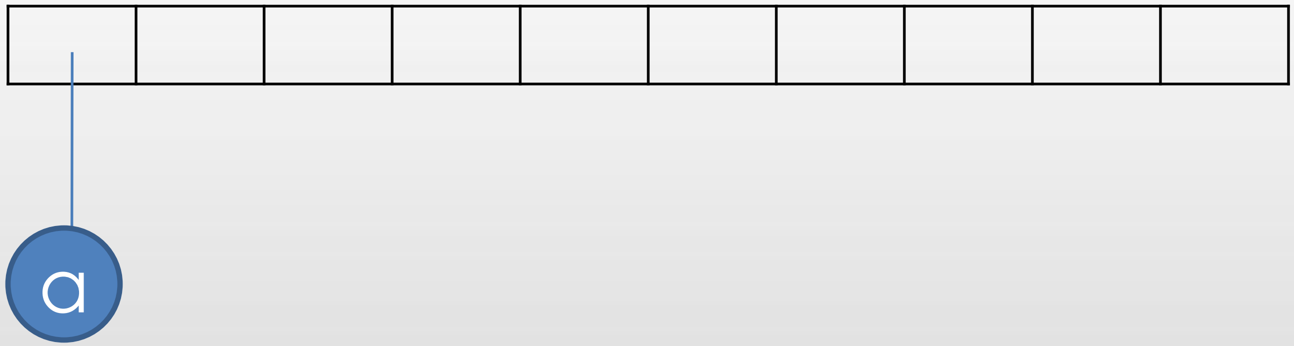
3. If the symbol is an operator, create a one-node tree out of the operator.
   - Pop the stack and make the right child of the operator node point to the binary tree just popped.
   - Pop the stack again and make the left child of the operator point to the binary tree just popped.
   - Push into the stack a pointer to the operator node.

Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-

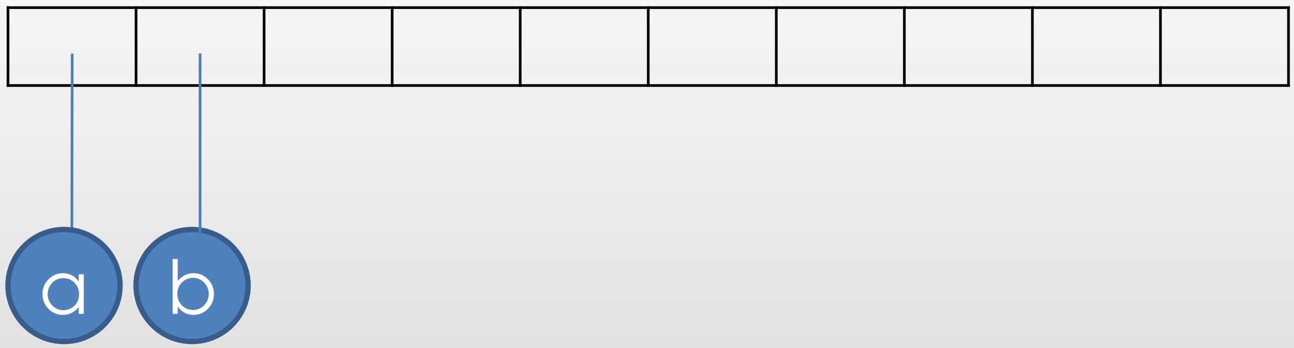
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



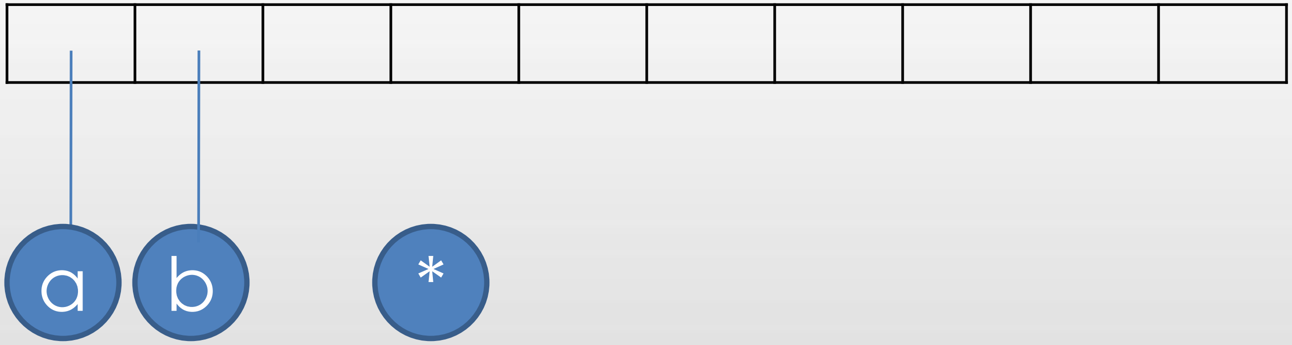
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



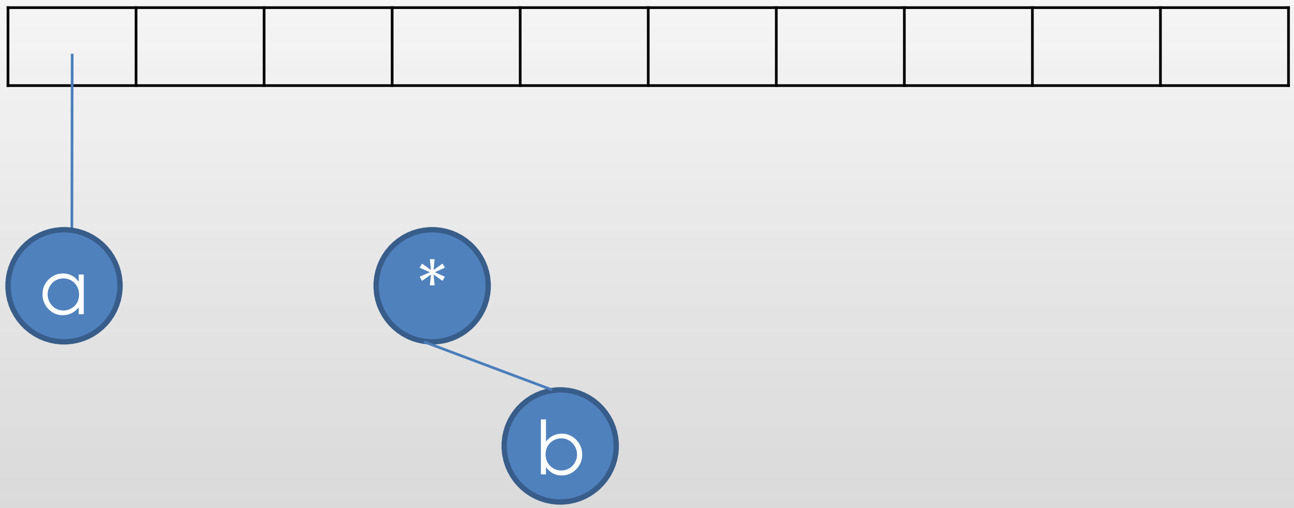
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



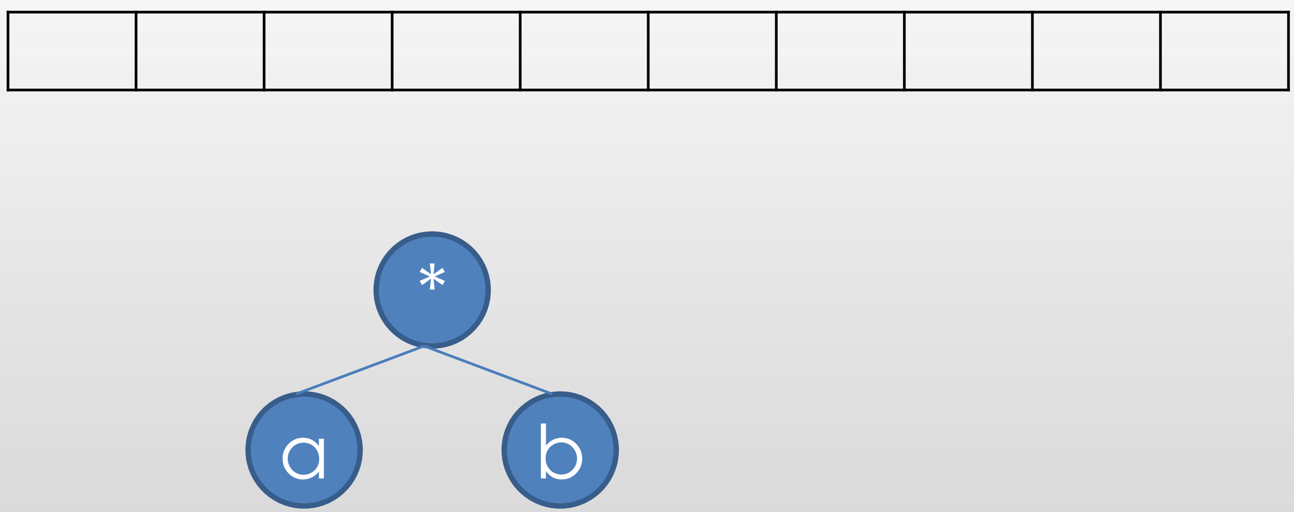
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



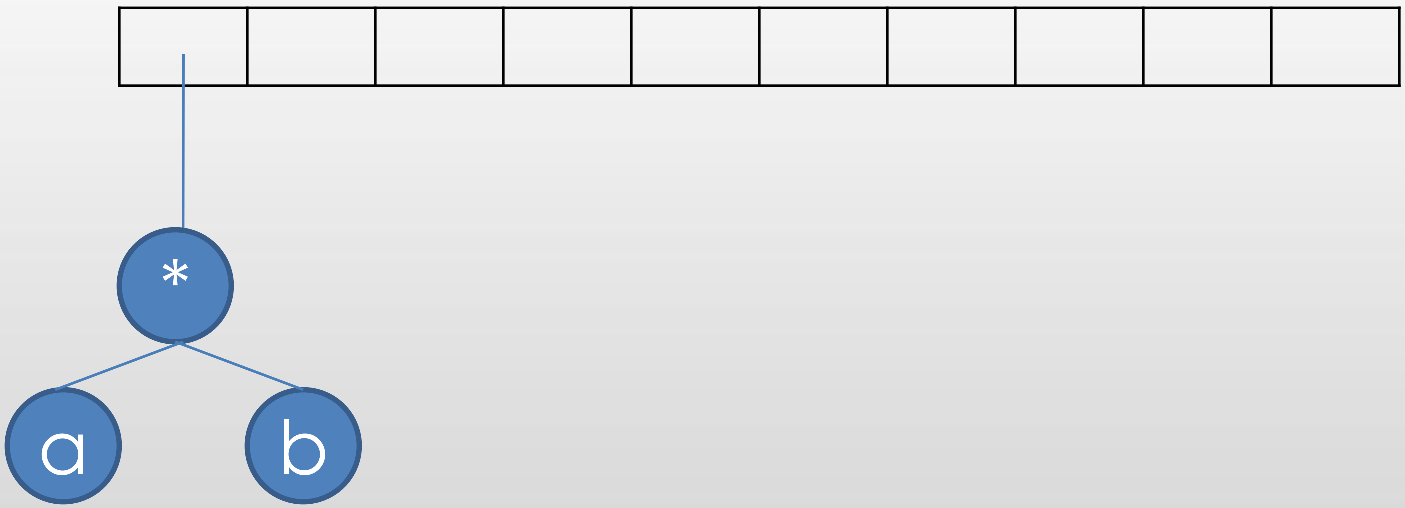
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



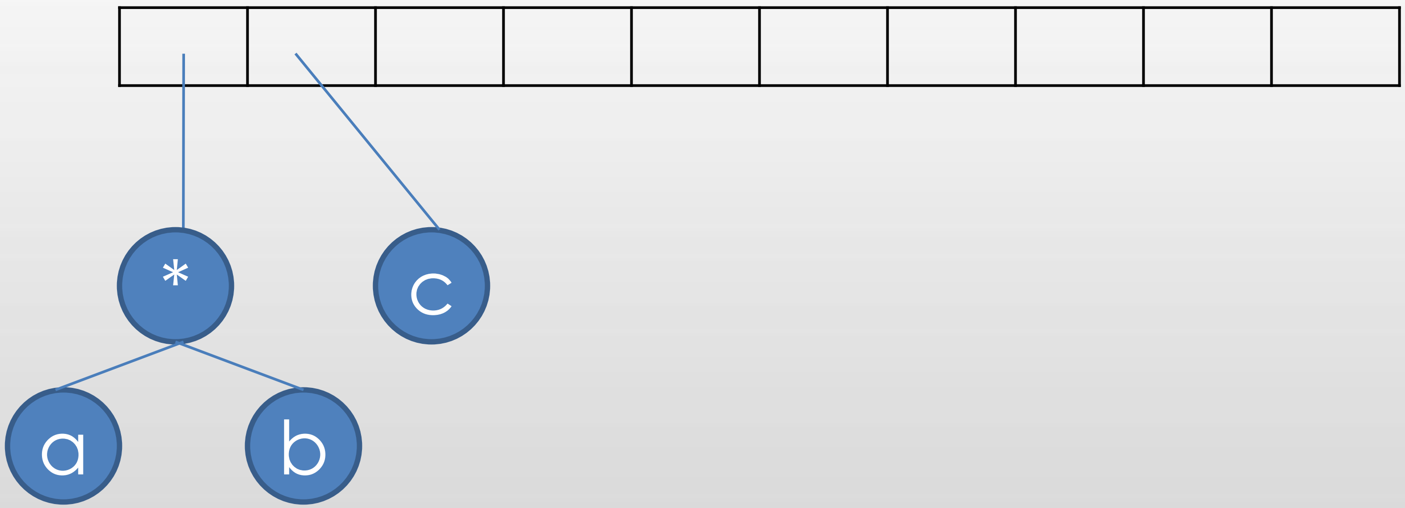
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



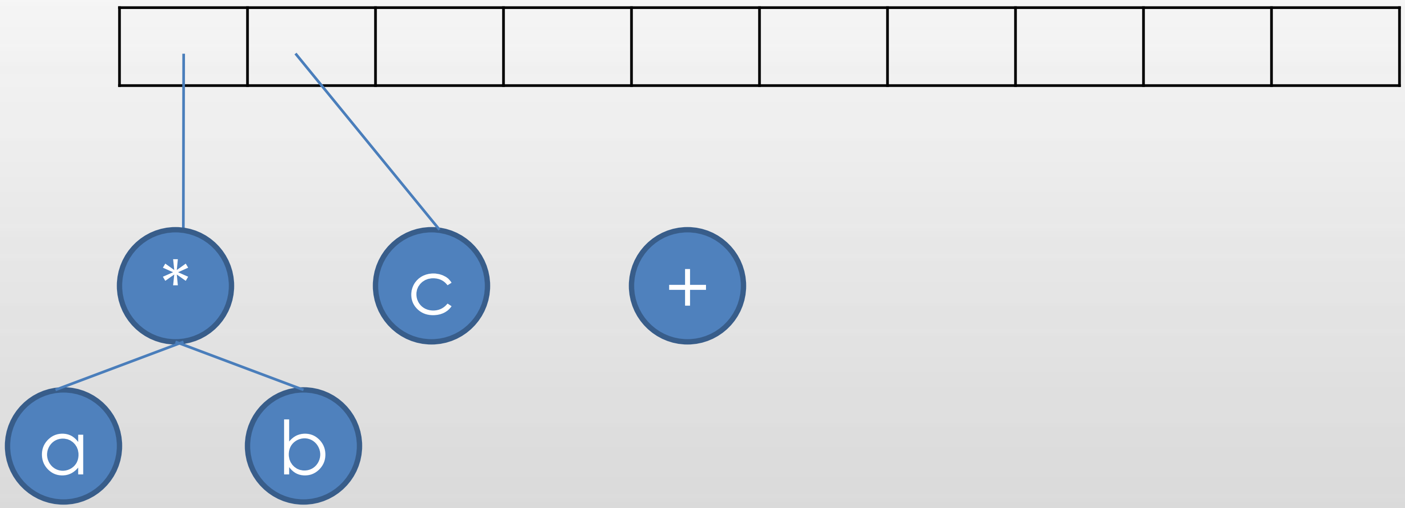
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



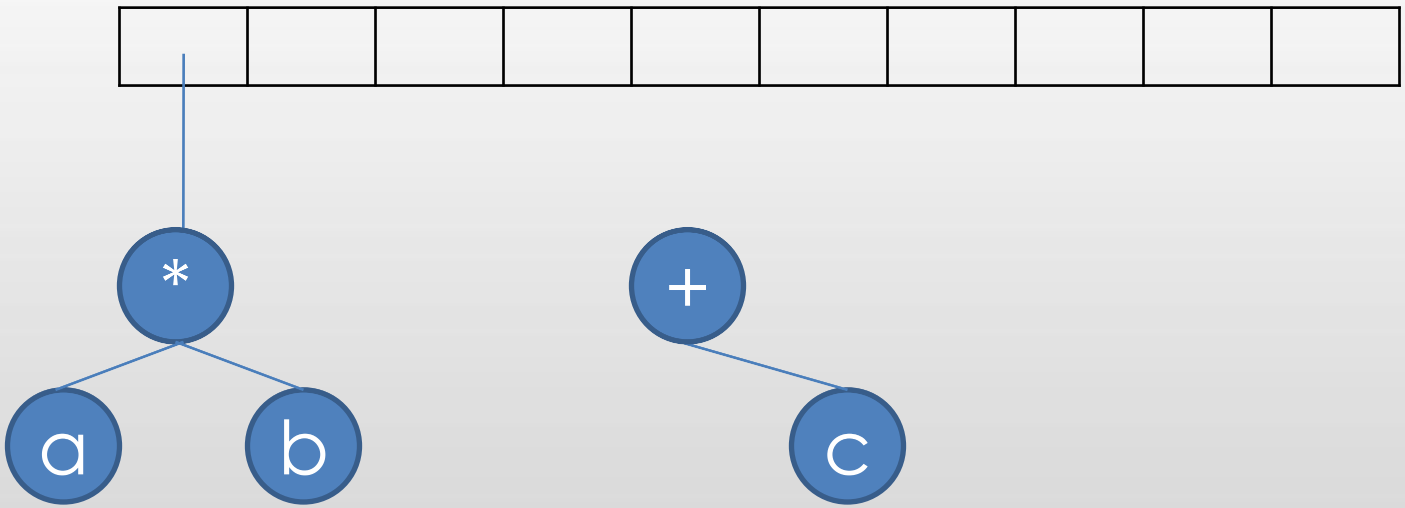
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



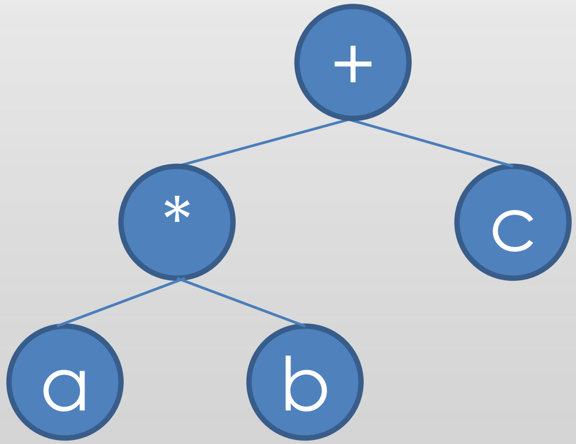
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



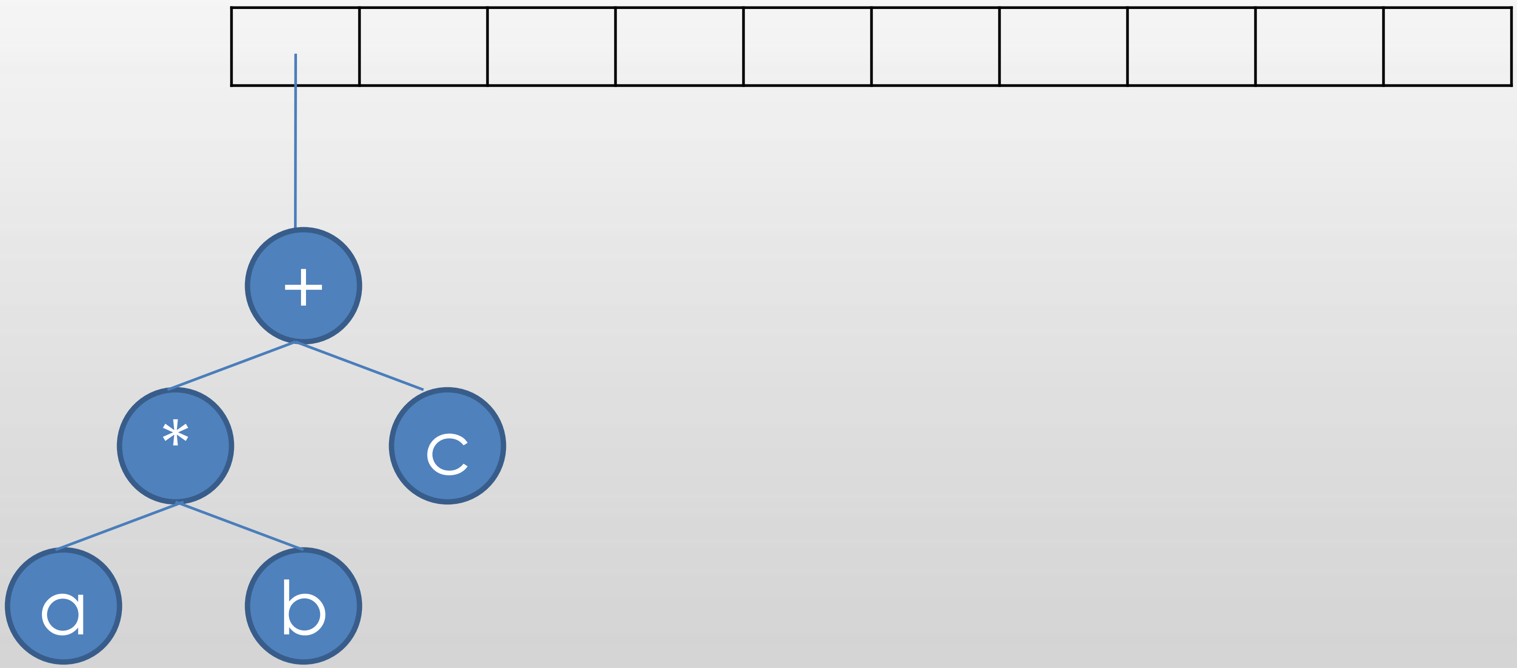
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



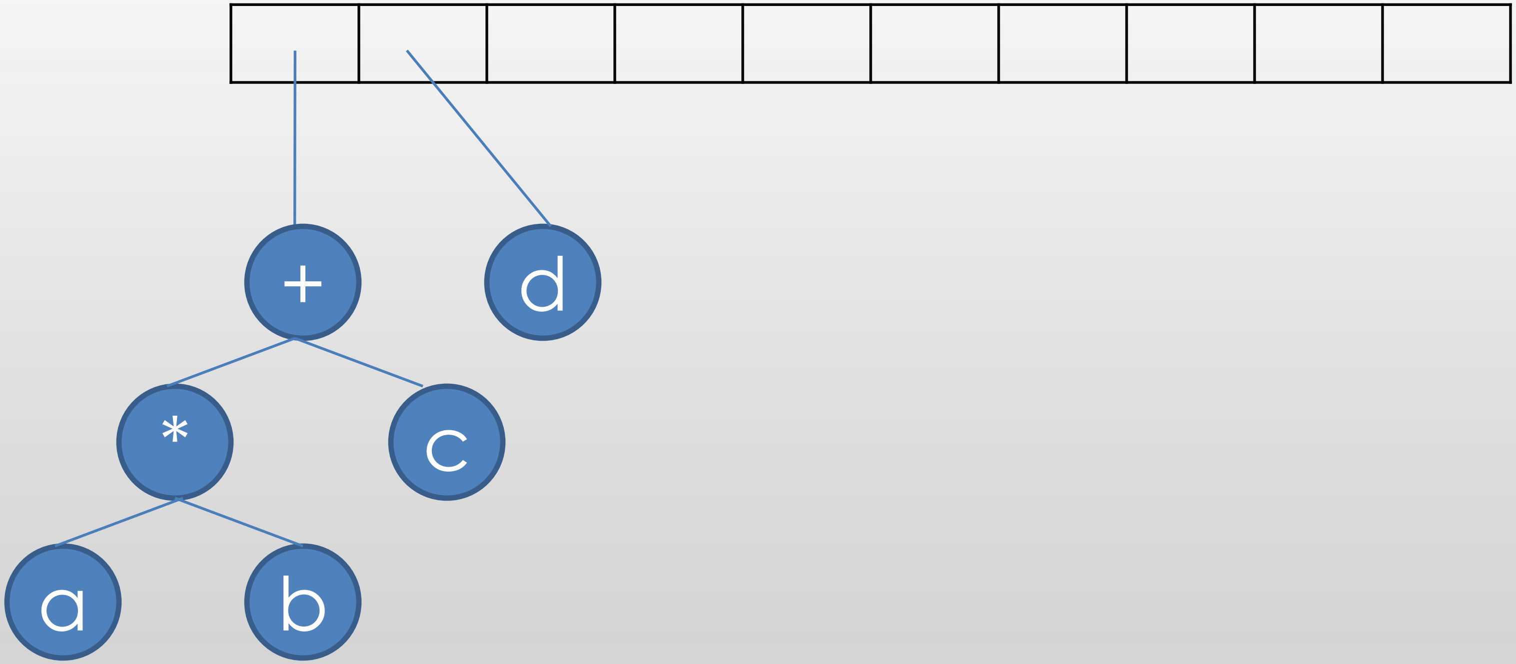
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



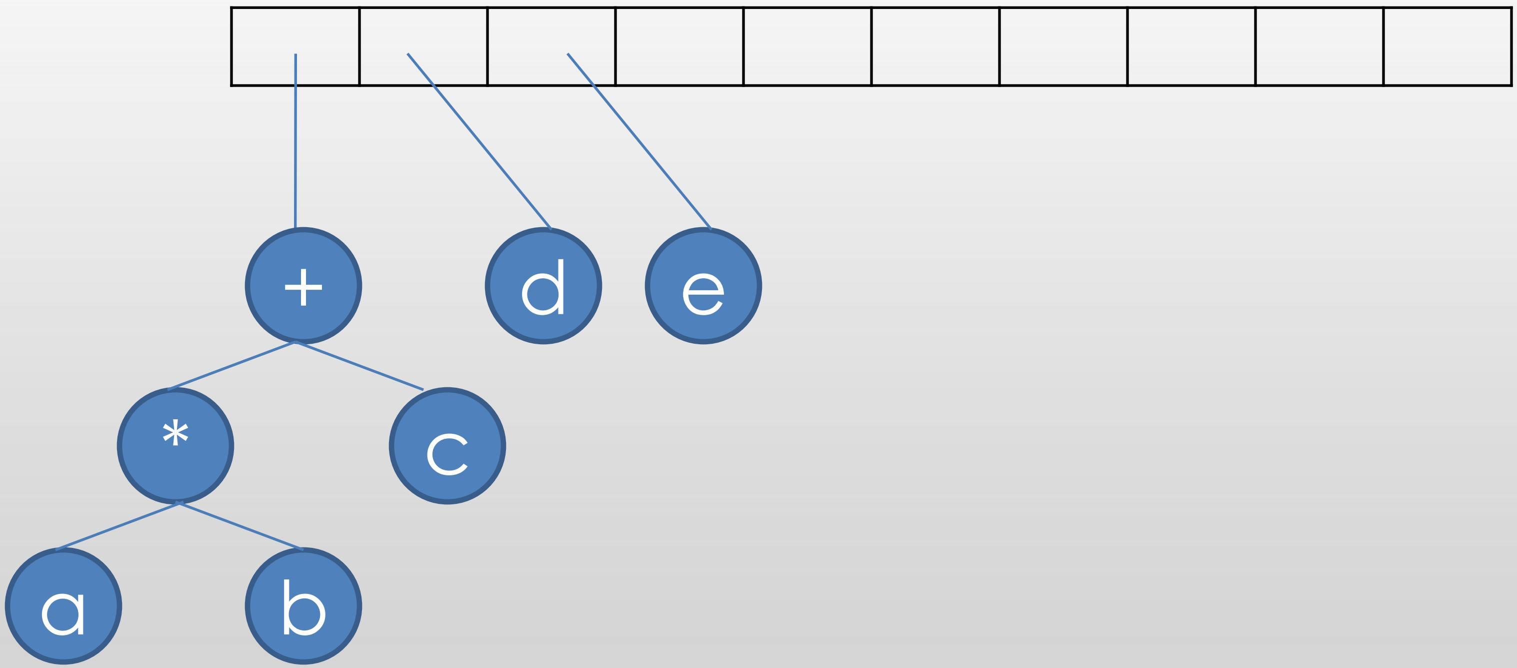
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



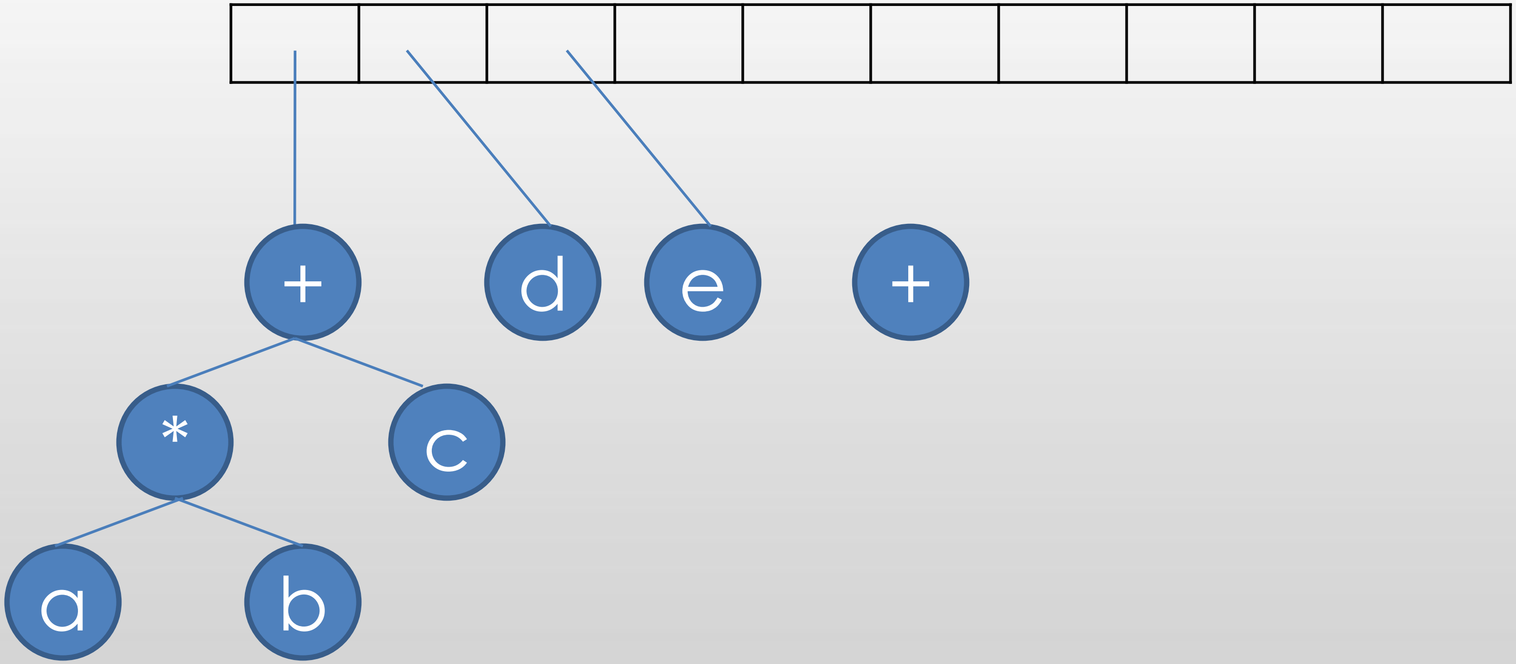
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



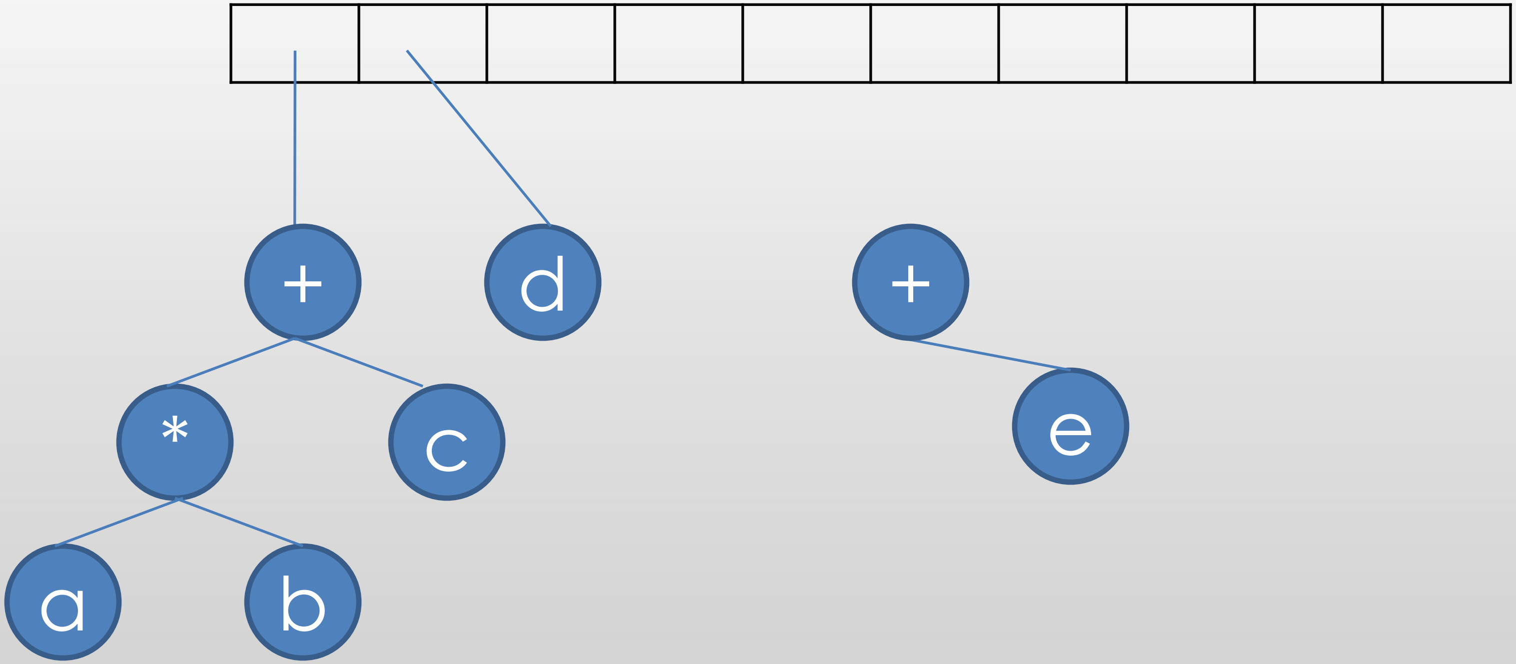
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



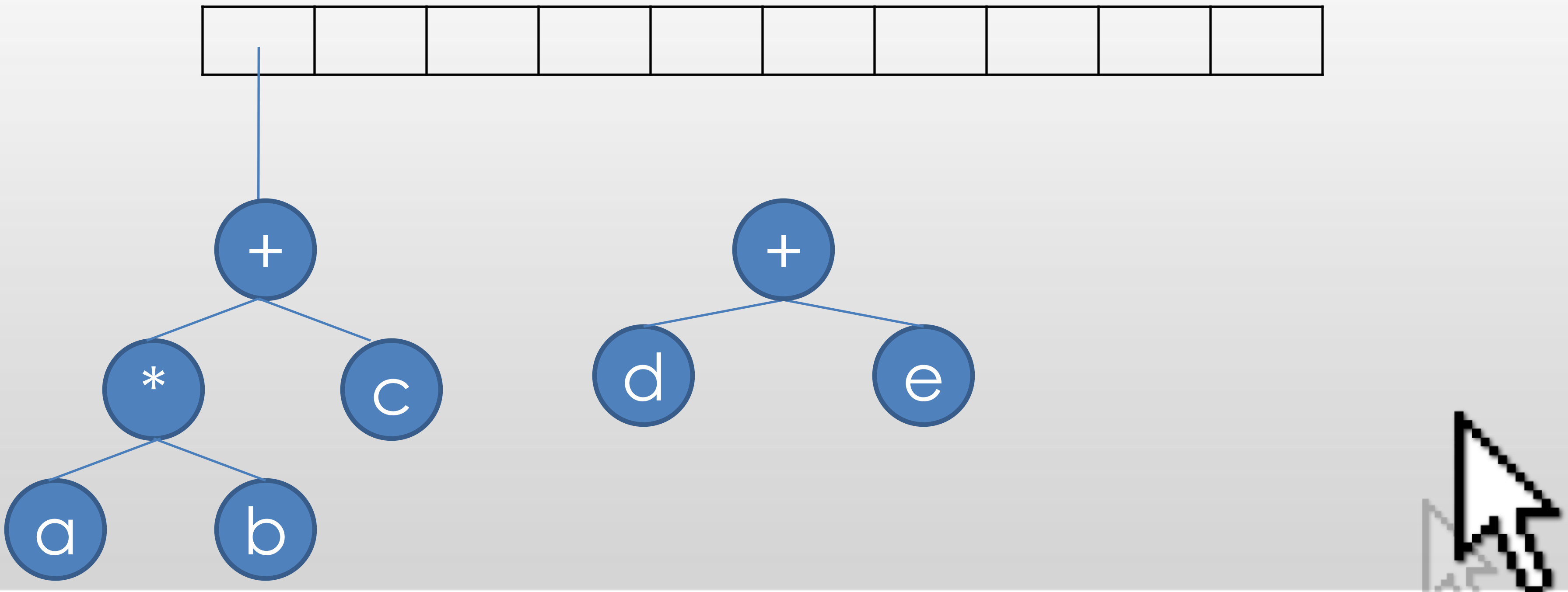
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



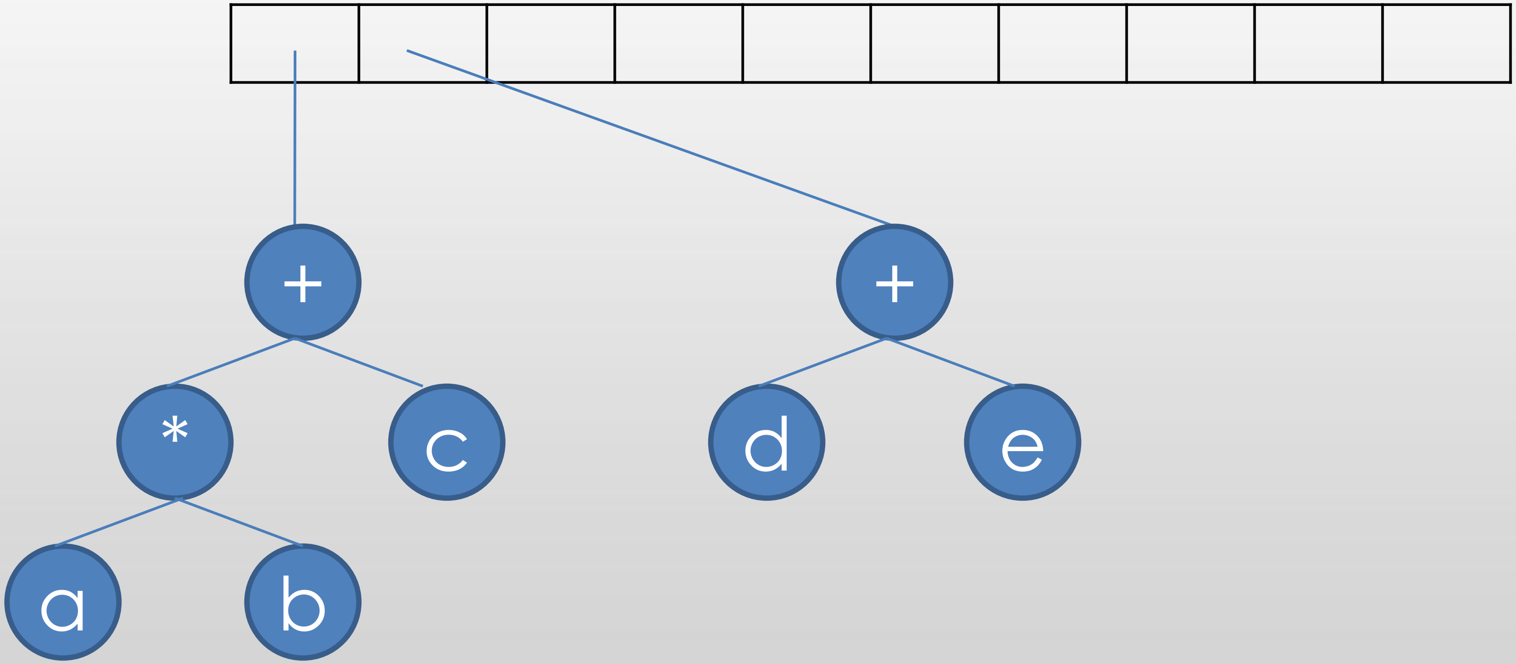
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



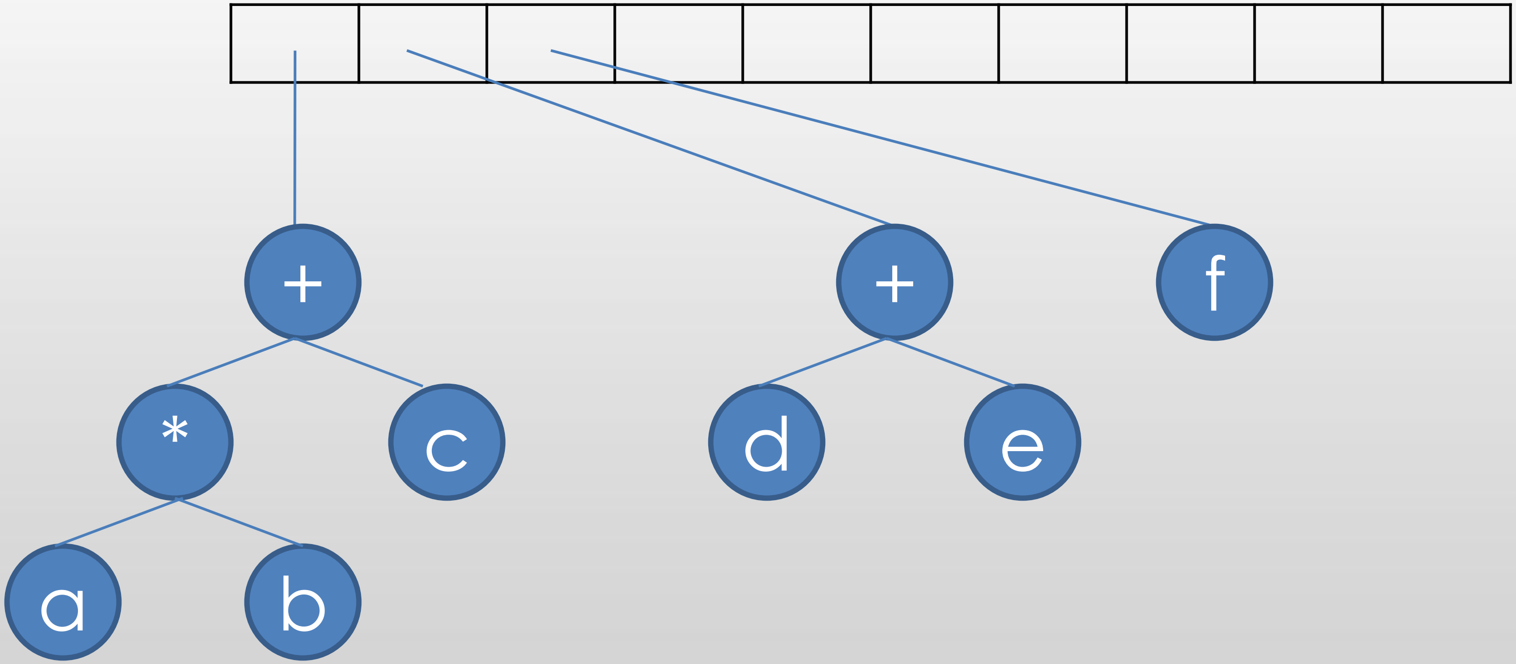
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



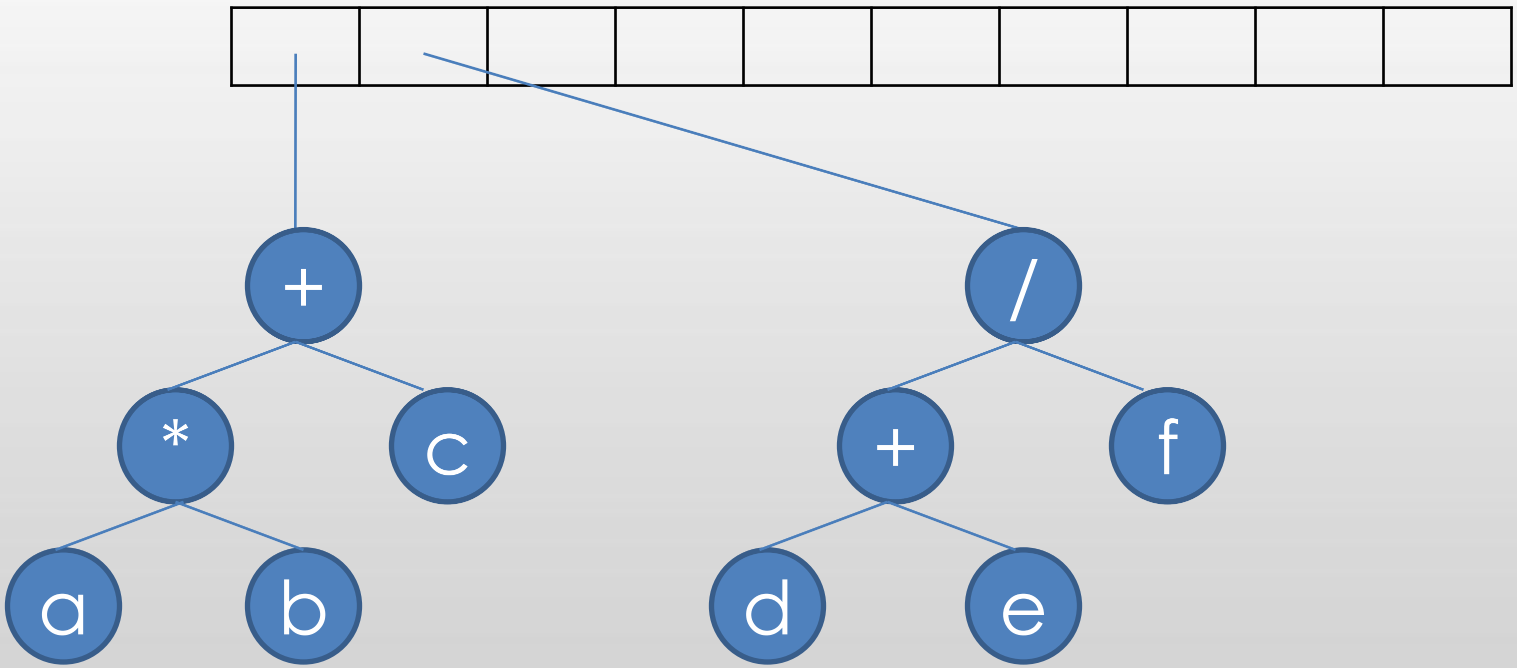
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



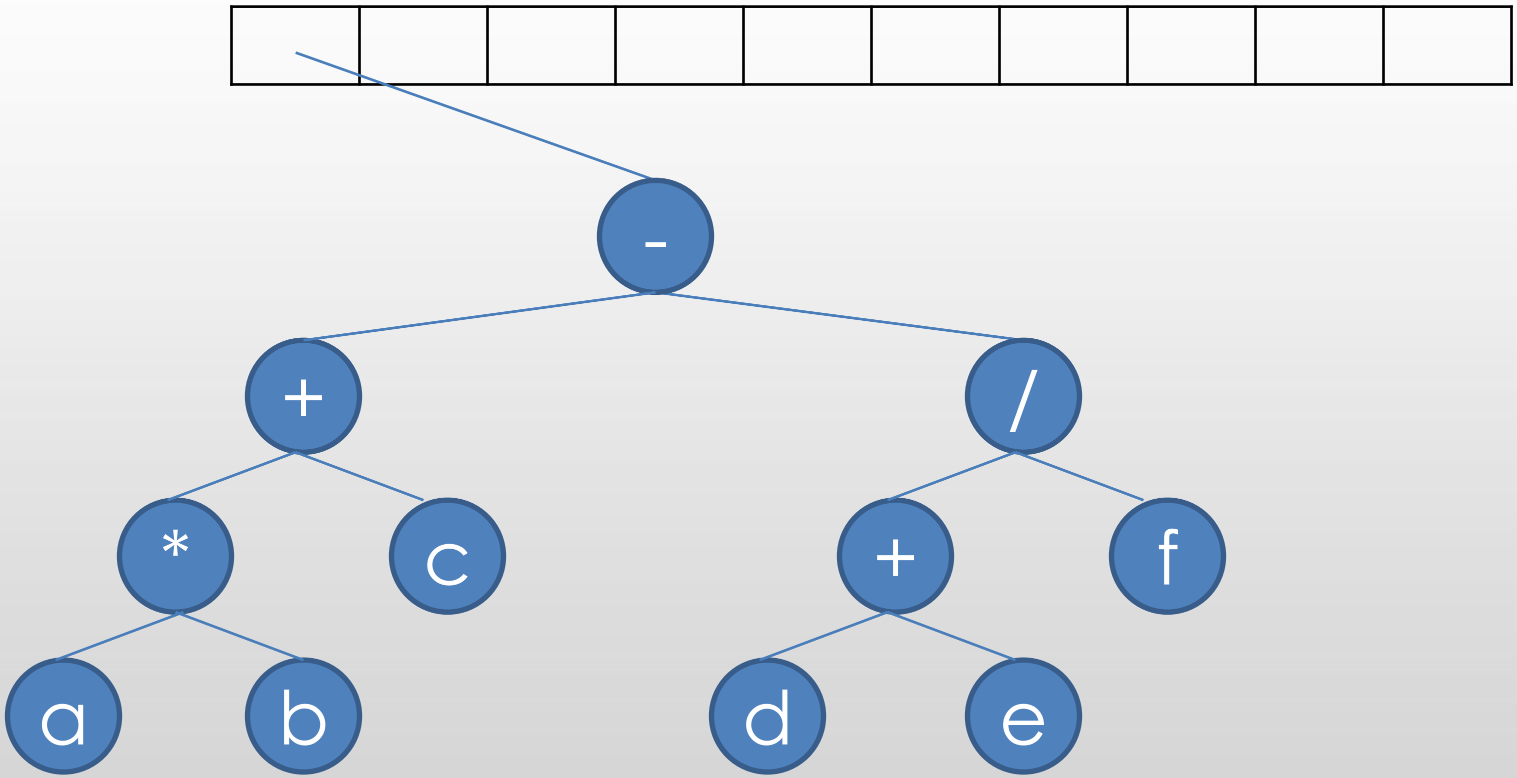
Expression Trees

- Infix Expression: (a*b+c)-((d+e)/f)
- Postfix Form: ab*c+de+f/-



Expression Trees

- Postfix Form: ab*c+de+f/-



4. Trees

4.3 Binary Search Trees

Binary Search Tree (BST)

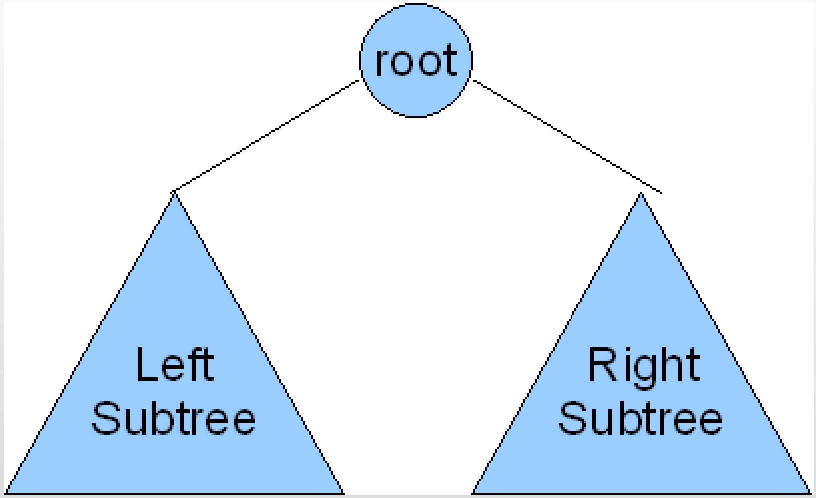
Definition: In a BST, the value of every node is greater than the values of the nodes in its left subtree and is less than the values of the nodes in its right subtree.

Operations:

- **search**, **insert**, **delete**
- **minimum**, **maximum**
- **successor**, **predecessor** (the value immediately greater and lesser than a node respectively)

BST

- Search order/tree property – i.e. Keys are organized in a special way, such that ...

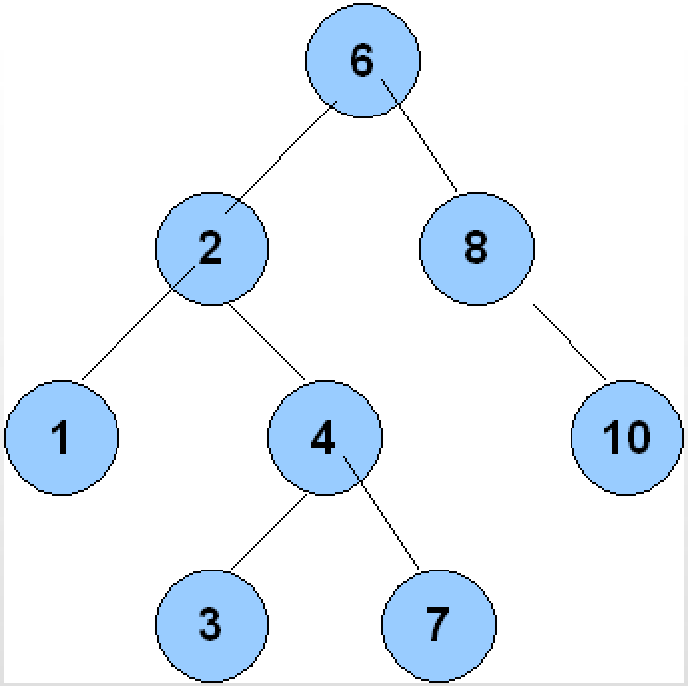


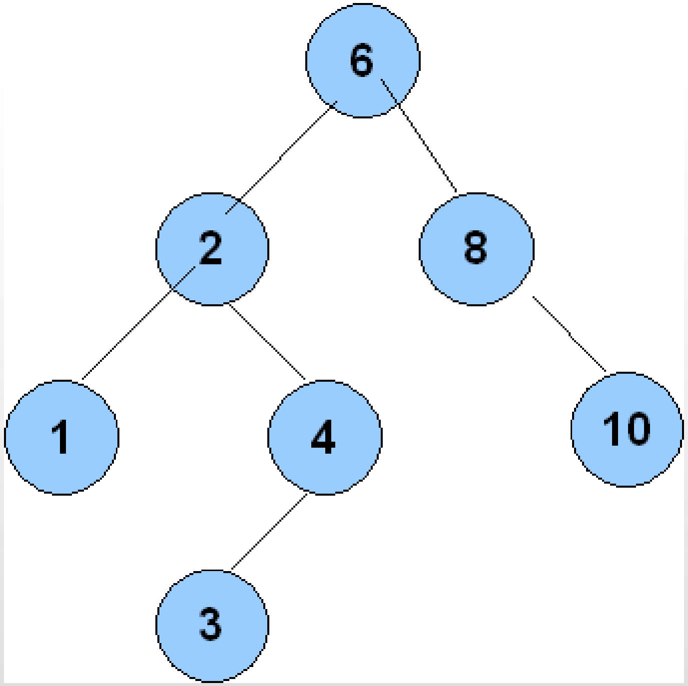
Two versions:

a) keys are distinct

b) duplicates are allowed

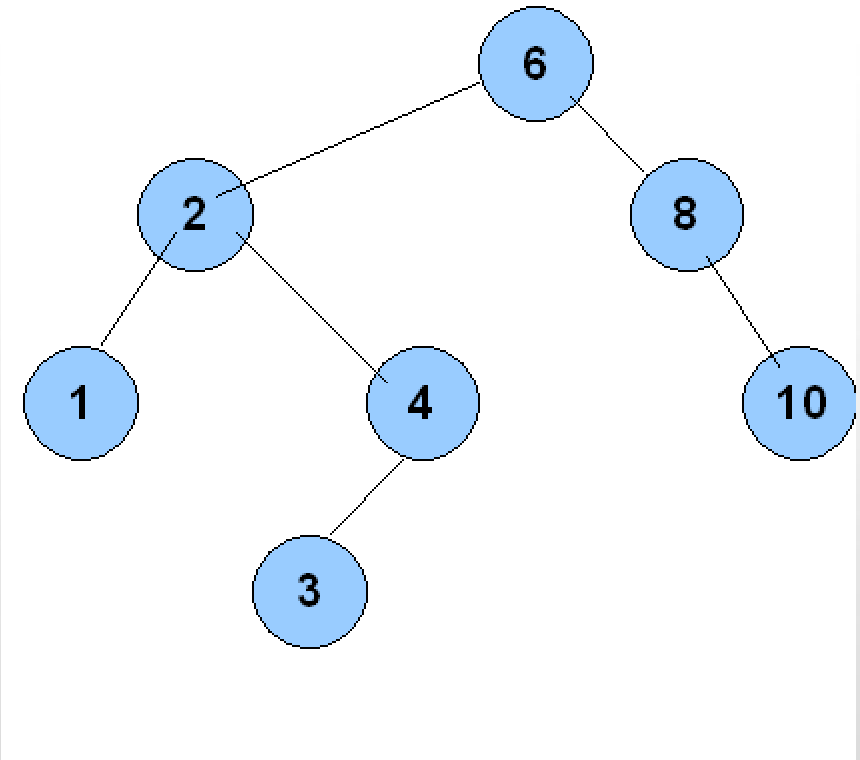
BST vs non-BST





Which is (not) a BST?

Search for a key



Two implementations:

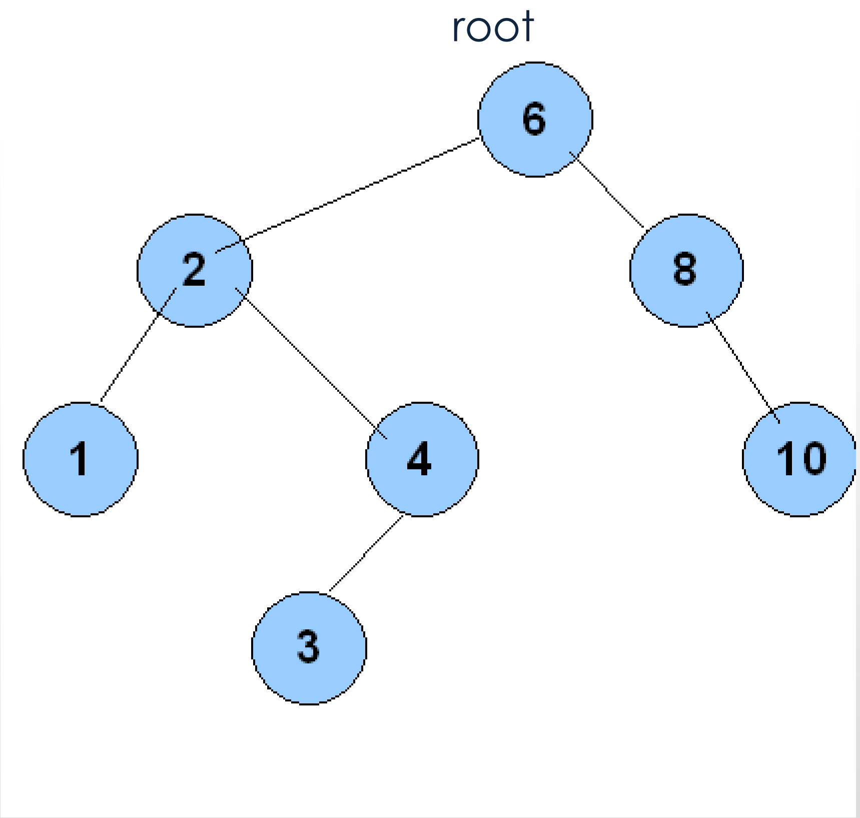
- Non-recursive
- Recursive

Search for a key

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;
```

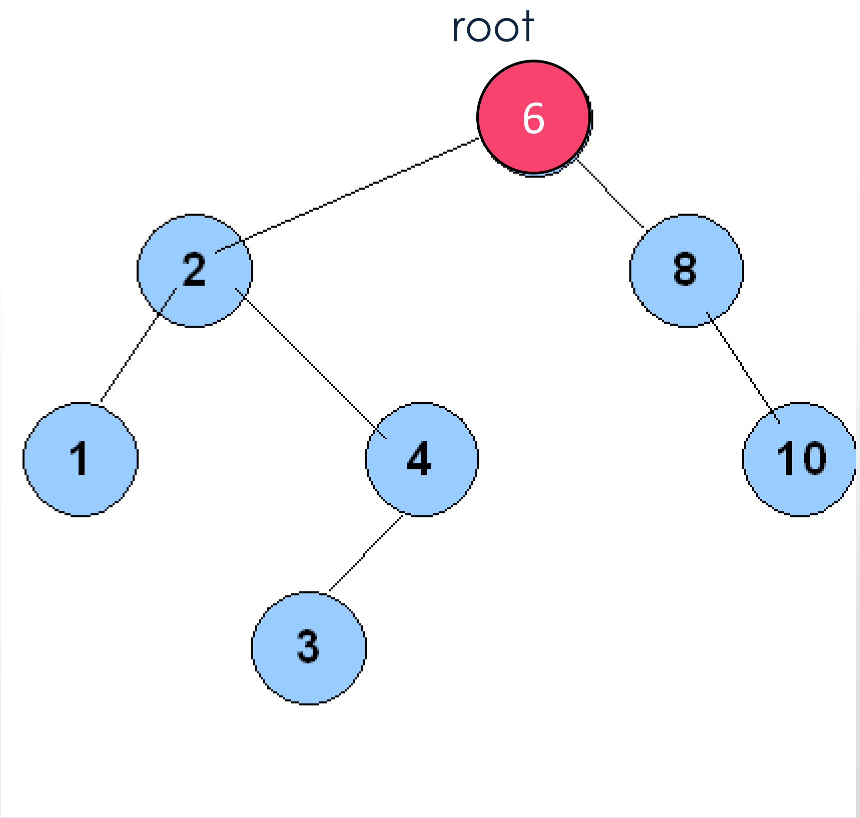
```
BST *search(BST *root, int x){
 BST *temp=root;

 while((temp!=NULL)&&(temp->value!=x)){
   if(x < temp->value)
     temp = temp->left;
   else
     temp = temp->right;
 }
 return temp;
}
```

Search for a key



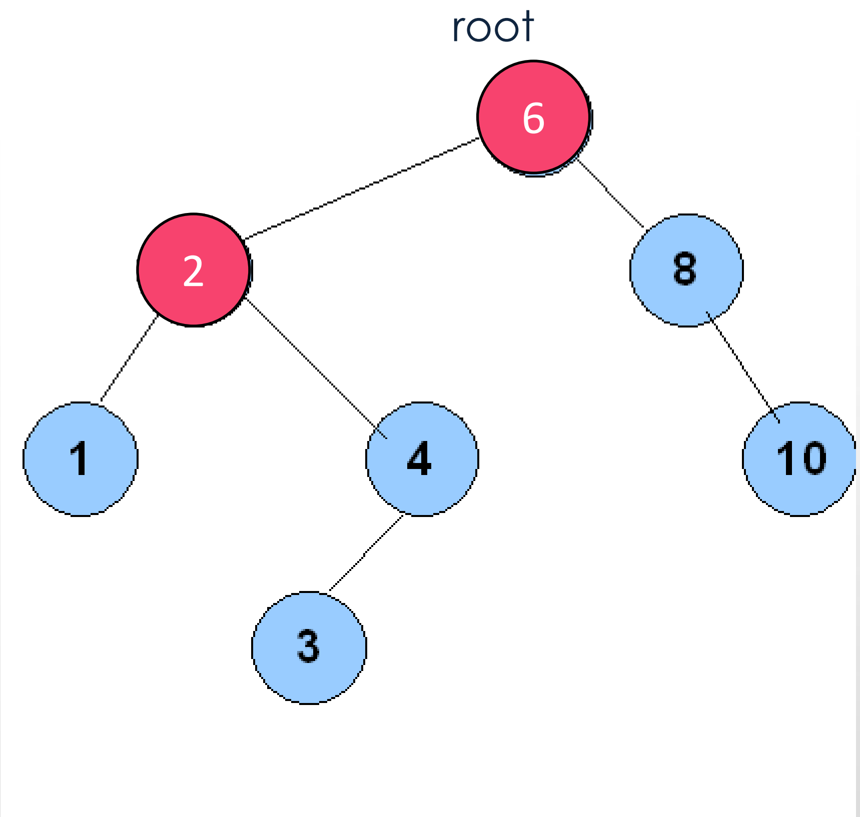
ptr = search(root, 3);

Search for a key



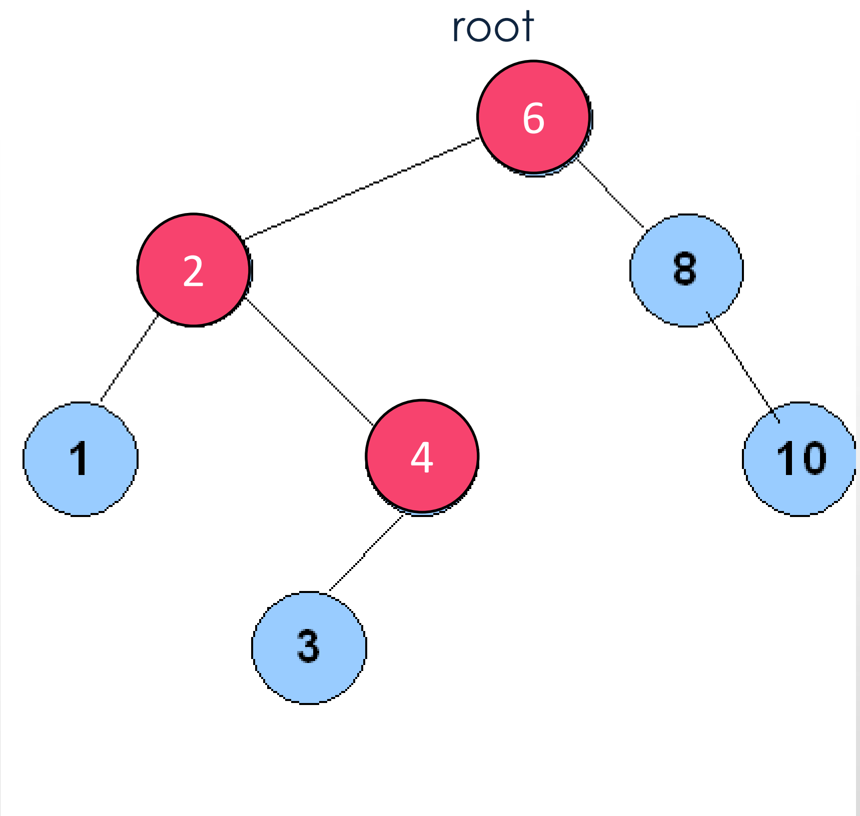
ptr = search(root, 3);

Search for a key



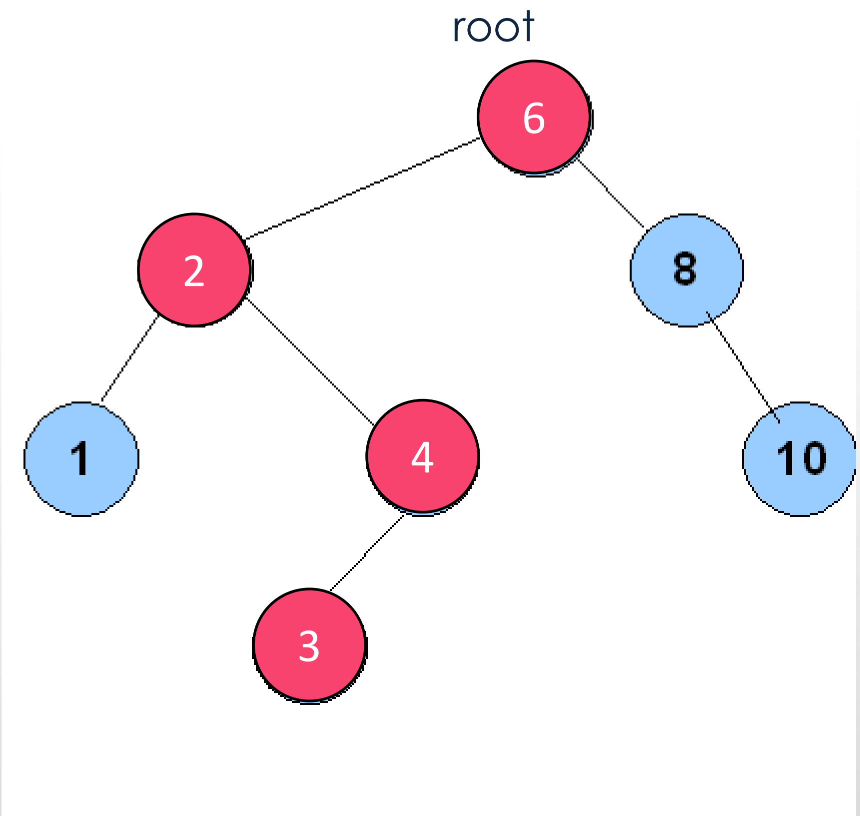
ptr = search(root, 3);

Search for a key



ptr = search(root, 3);

Search for a key



ptr = search(root, 3);

Search for a key

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;
```

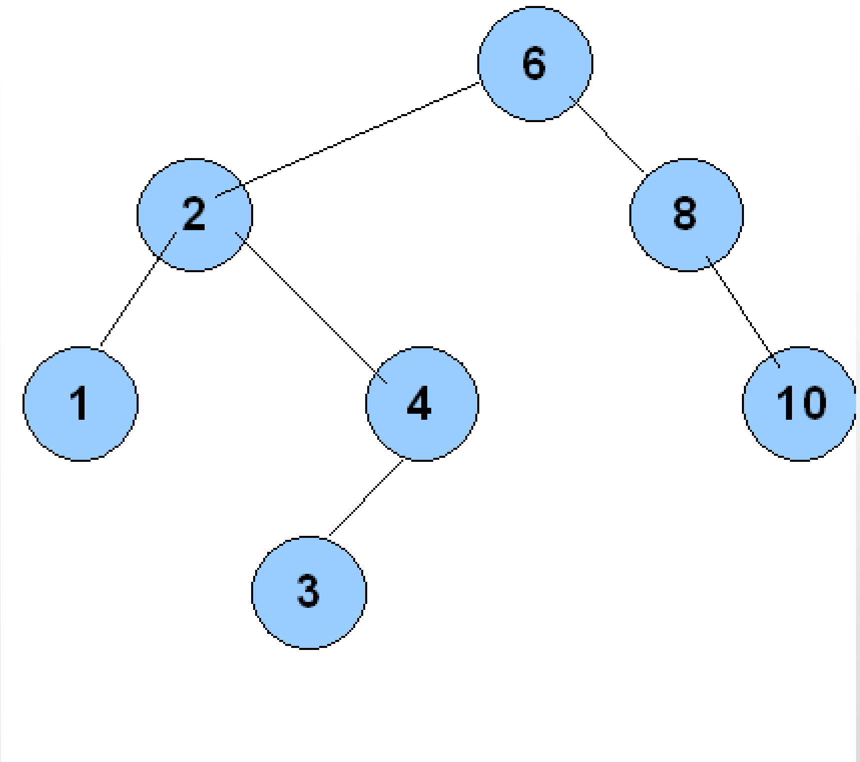
```
BST *search(BST *root, int x){

  if(root==NULL)
    return NULL;
  if(x < root->value)
    return(search(root->left, x));
  if(x > root->value)
    return(search(root->right, x));
  else
    return root;
}
```

Find minimum

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;
```

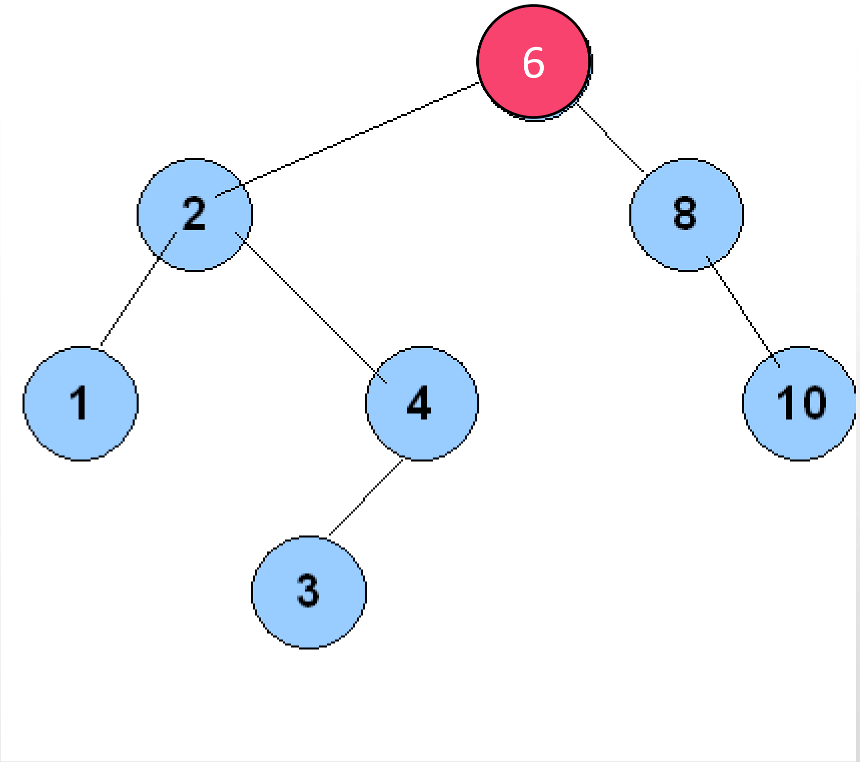
```
BST *find_min(BST *root){
  BST *temp=root;

  if(temp!=NULL){
    while(temp->left!=NULL)
       temp = temp->left;
    return temp;
  }
  else
    return NULL;
}
```

Find Minimum



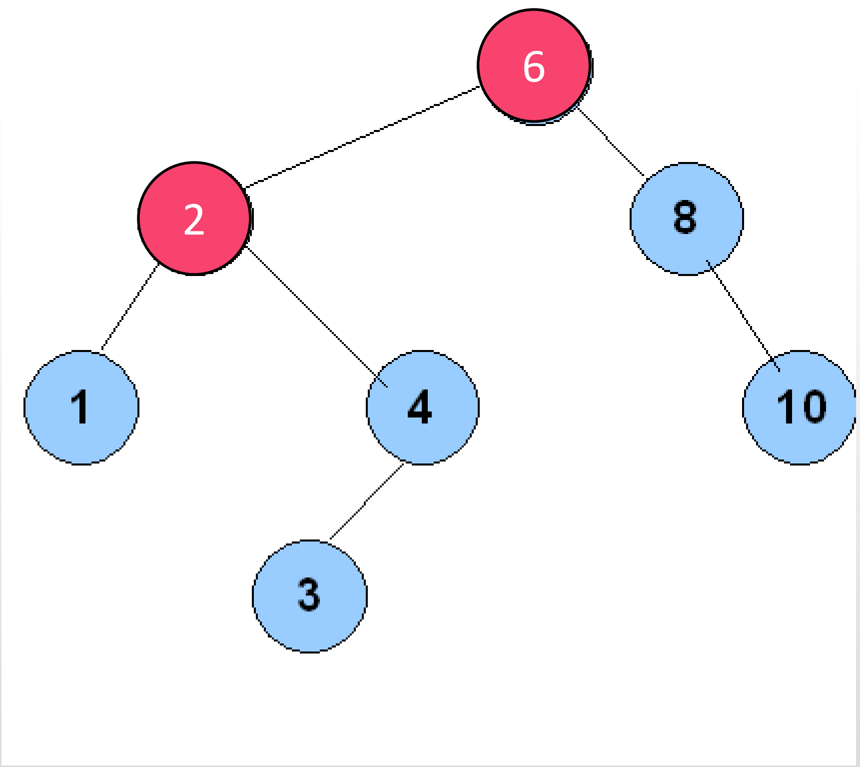
ptr = find_min(root);

Find Minimum



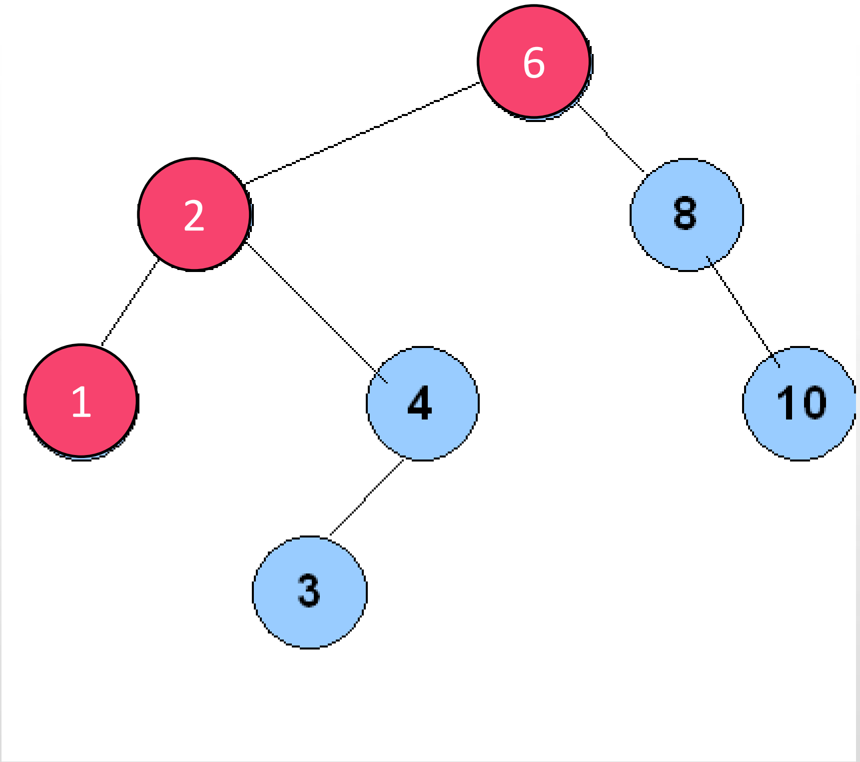
ptr = find_min(root);

Find Minimum



ptr = find_min(root);

Find Minimum



ptr = find_min(root);

Find minimum

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;
```

```
BST *find_min(BST *root){

  if(root==NULL)
    return NULL;
  else if(root->left==NULL)
    return(root);
  else
    return(find_min(root->left));
}
```

Insertion

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;

int main(){
  BST *root=NULL;

}
```

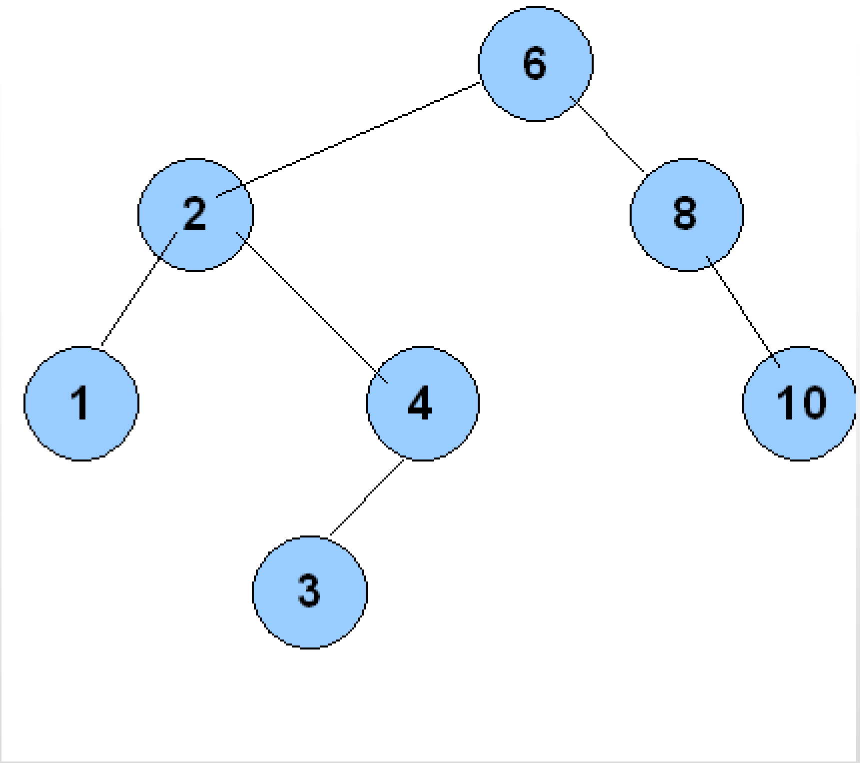
```
void insert(BST *root, int x){
  BST *temp;

  temp=(BST *)malloc(sizeof(BST));
  if(temp==NULL){
    printf("Insufficient Memory");
    exit(1);
  }
  temp->value=x;
  temp->left=NULL;
  temp->right=NULL;
  temp->parent=NULL;

  insert2(root, temp);
}
```

Insertion

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;

int main(){
  BST *root=NULL;

}
```

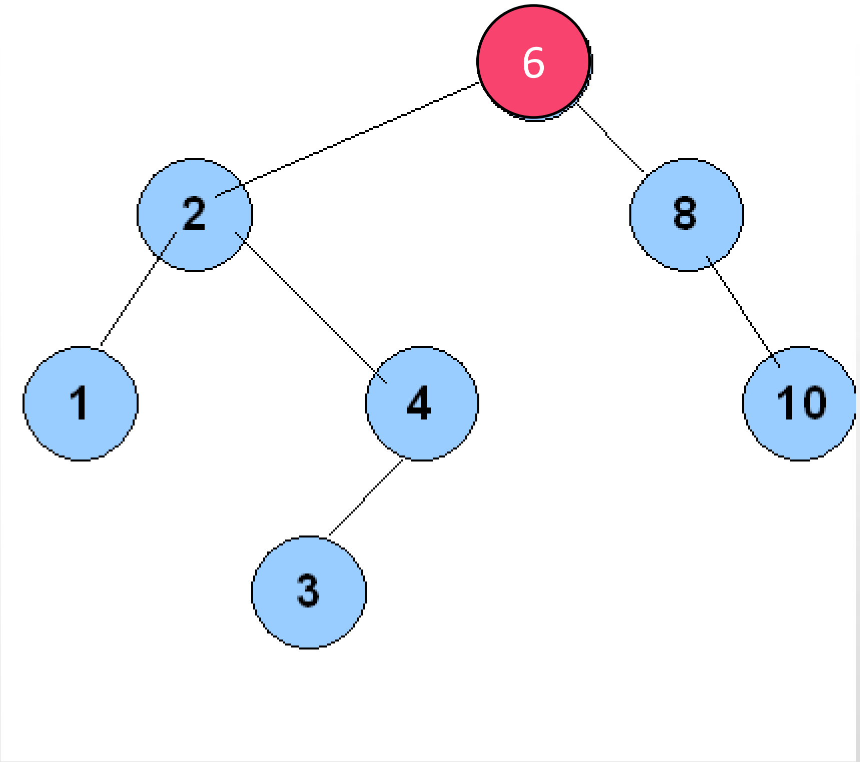
```
void insert2(BST *root, BST *temp){

 if(root==NULL)
   root=temp;
 else{
  temp->parent=root;
  if ((root)->value > temp->value)
   insert2(root->left,temp);
  else
   insert2(root->right,temp);
 }
}
```

Insertion



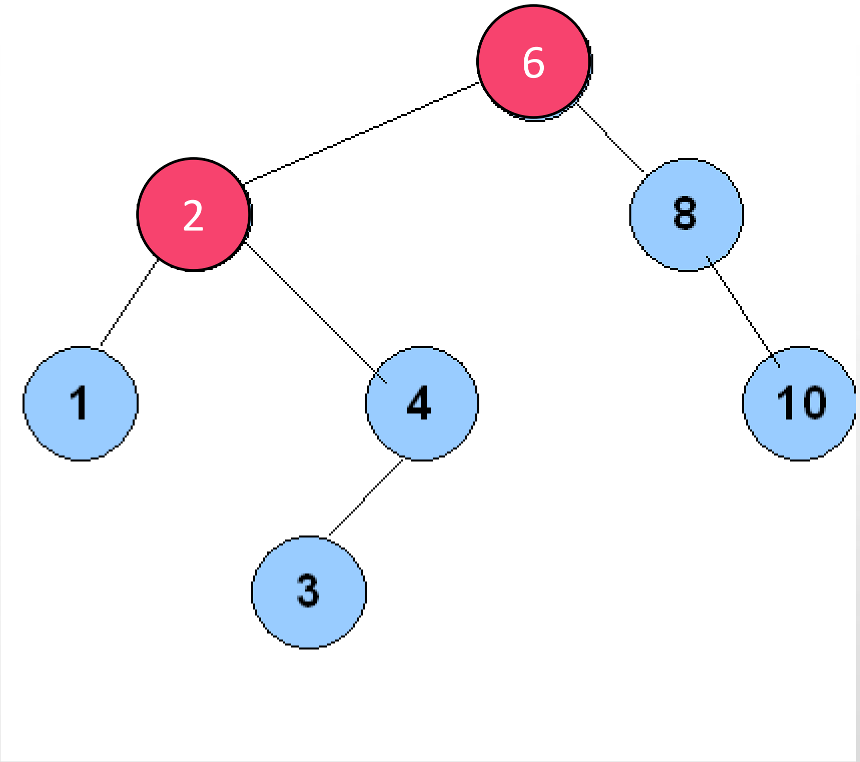
ptr = insert(root, 5);

Insertion



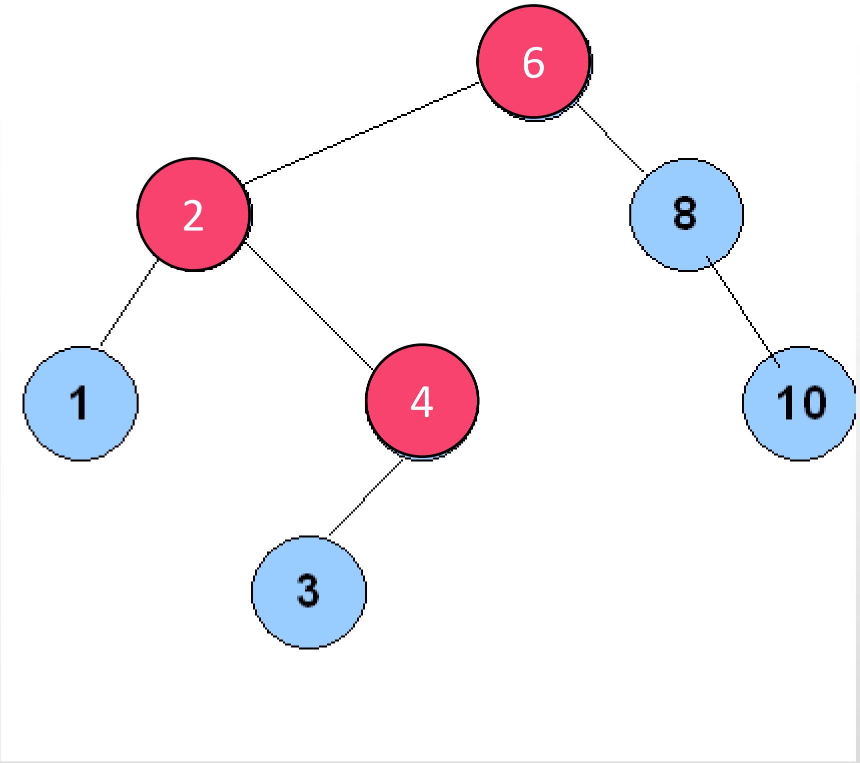
ptr = insert(root, 5);

Insertion



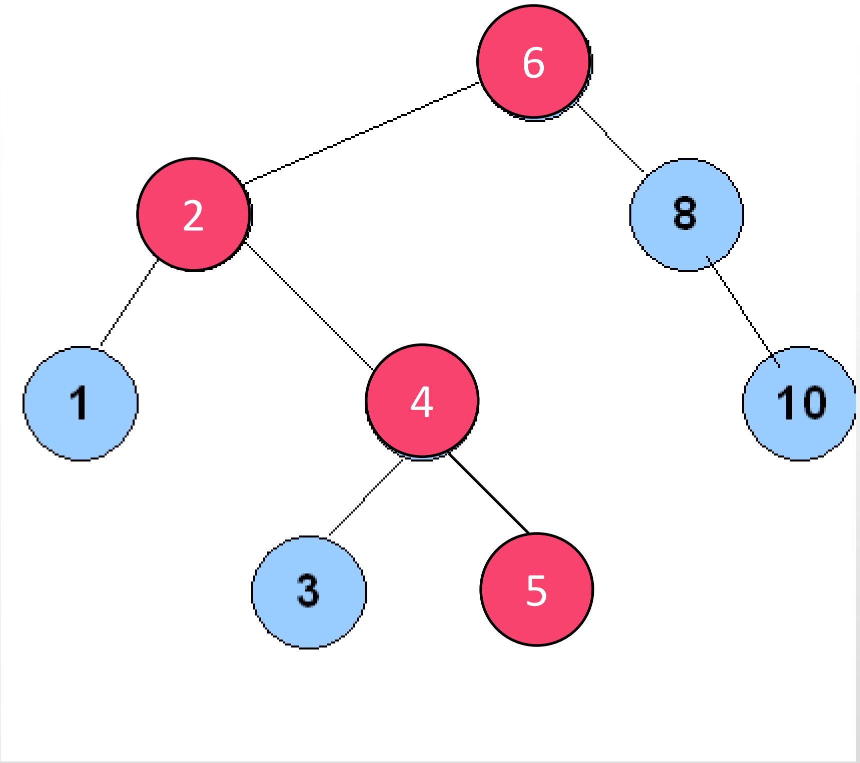
ptr = insert(root, 5);

Insertion



ptr = insert(root, 5);

Insertion

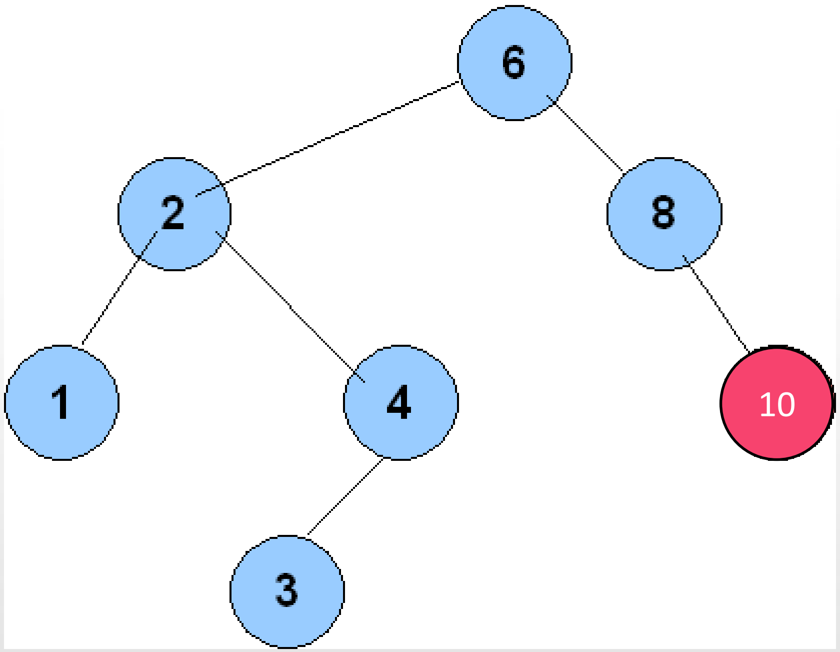


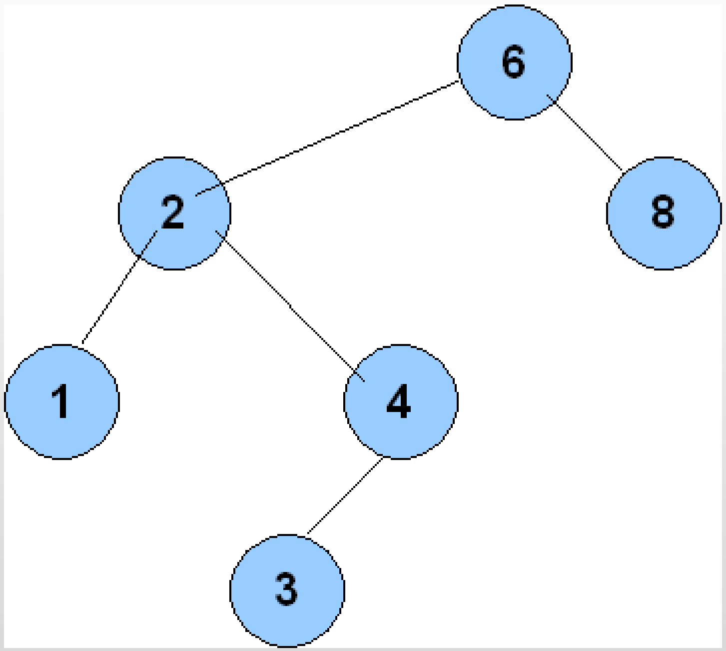
ptr = insert(root, 5);

Deletion

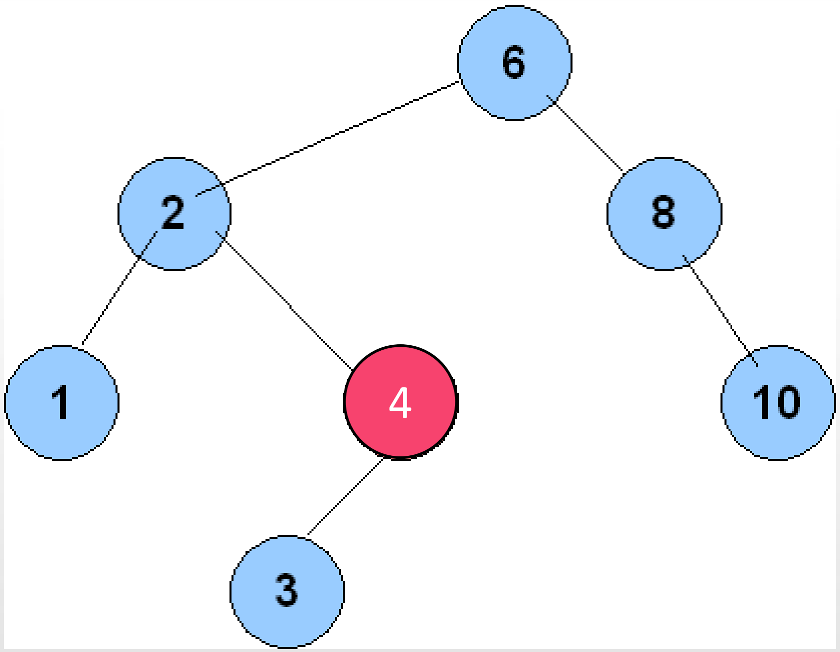
- 3 Cases
  - Node is a leaf
  - Node is non-leaf with 1 child
  - Node is non-leaf with 2 children

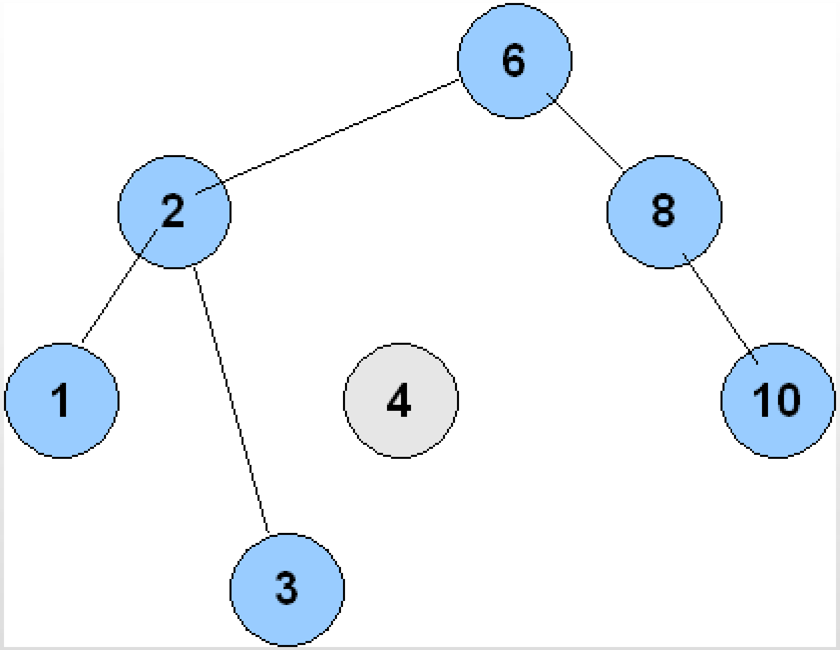
Deleting a leaf node



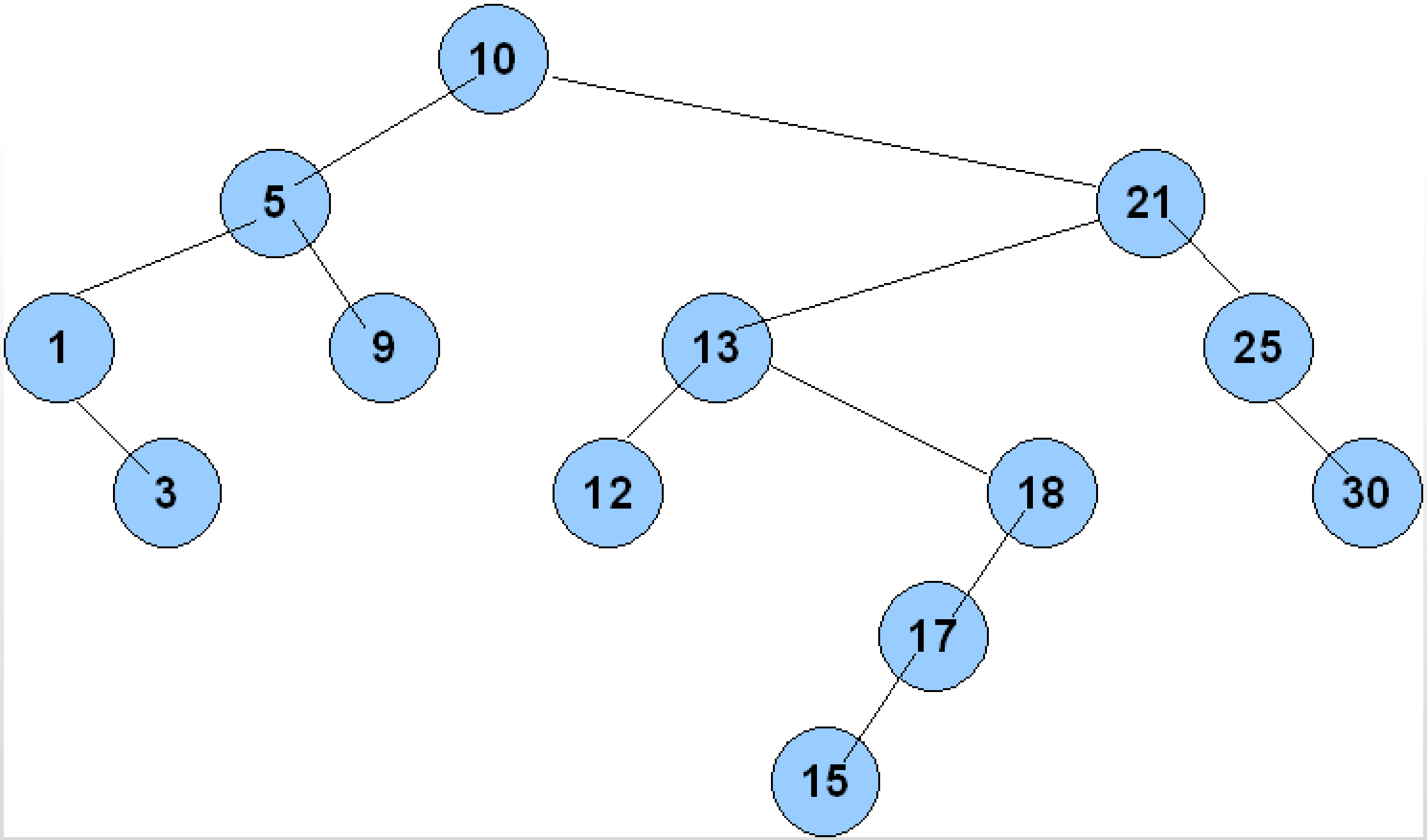


Deleting non-leaf node with 1 child



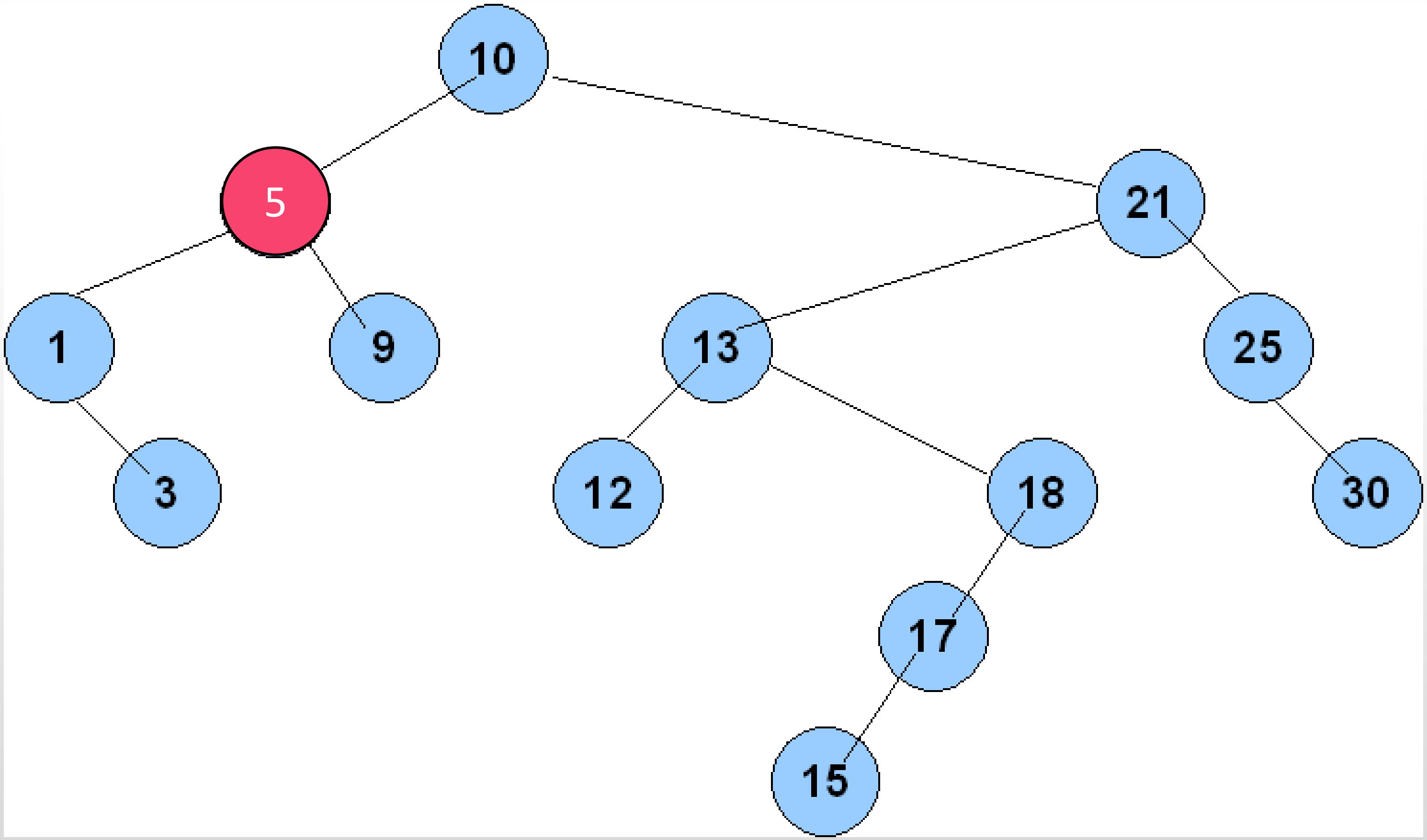


Deleting non-leaf node with 2 children



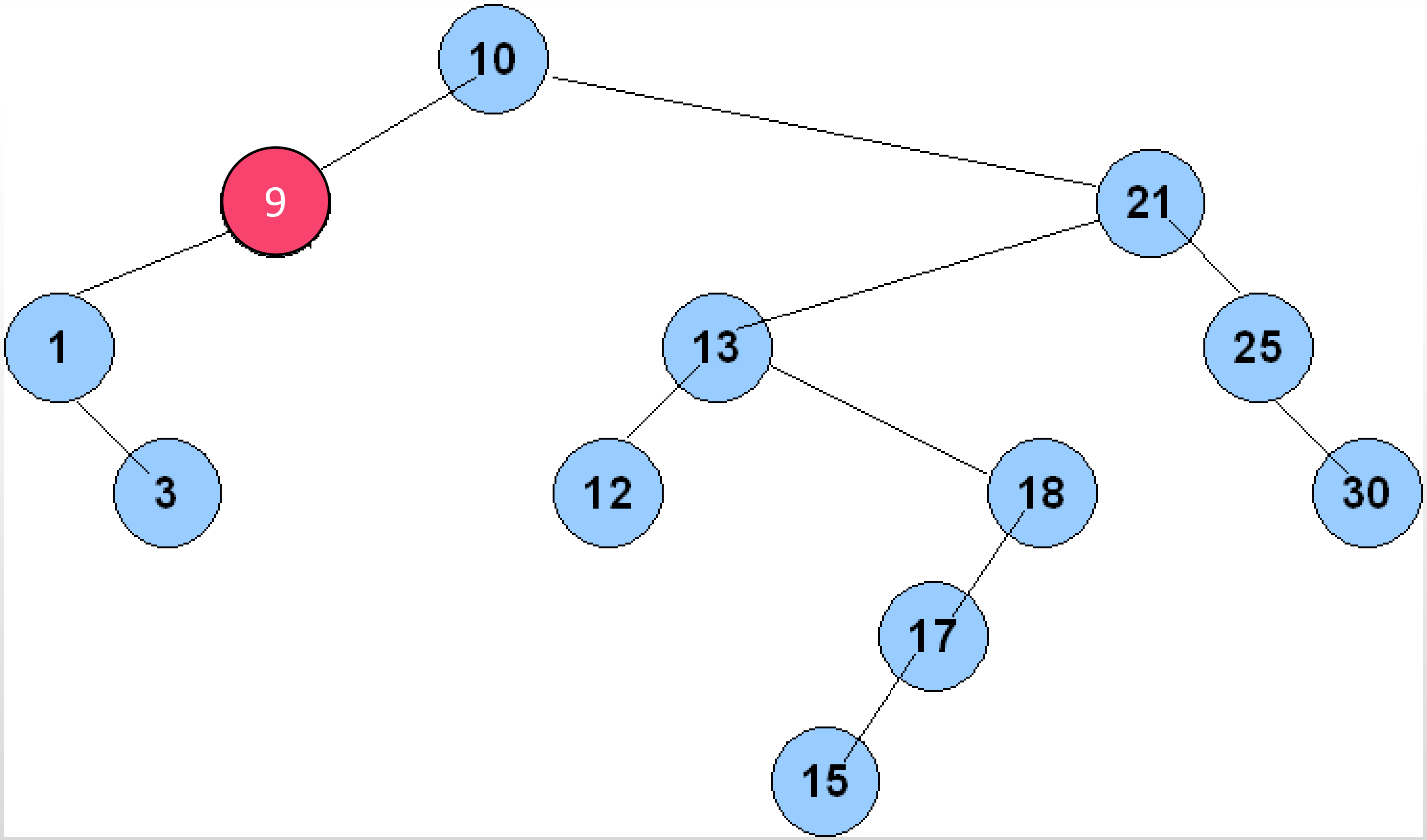
- Delete 5

Deleting non-leaf node with 2 children



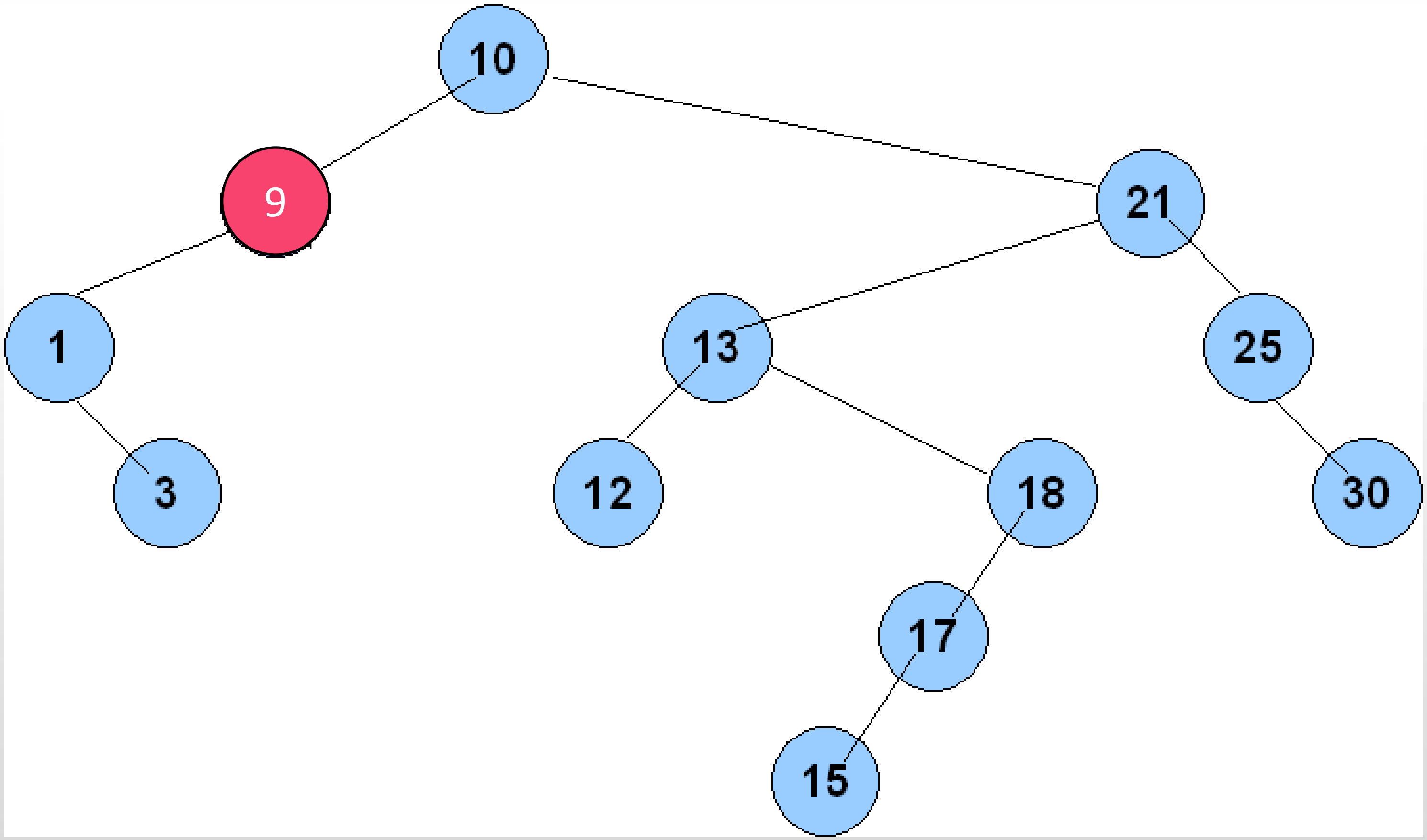
- Delete 5

Deleting non-leaf node with 2 children



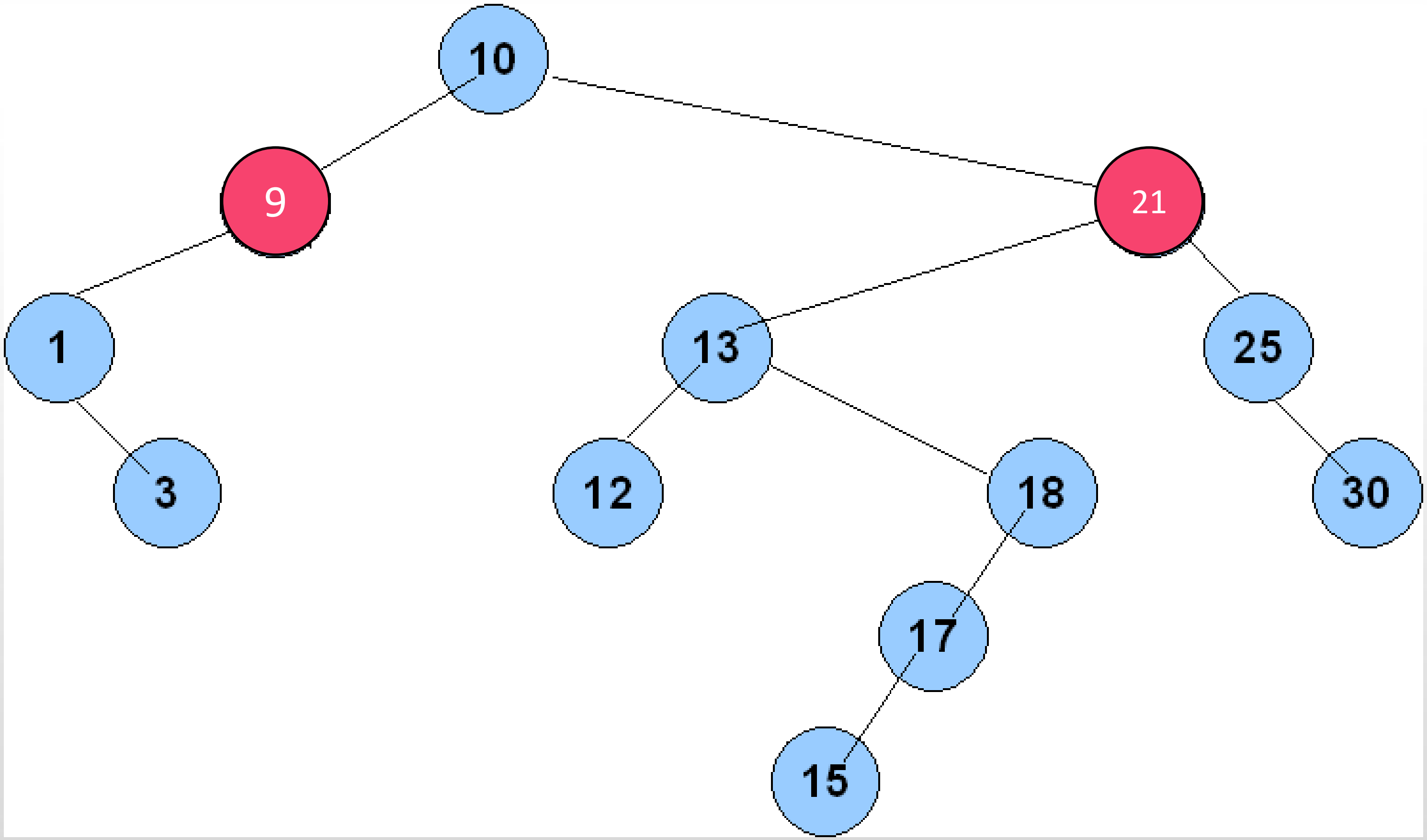
- Delete 5

Deleting non-leaf node with 2 children



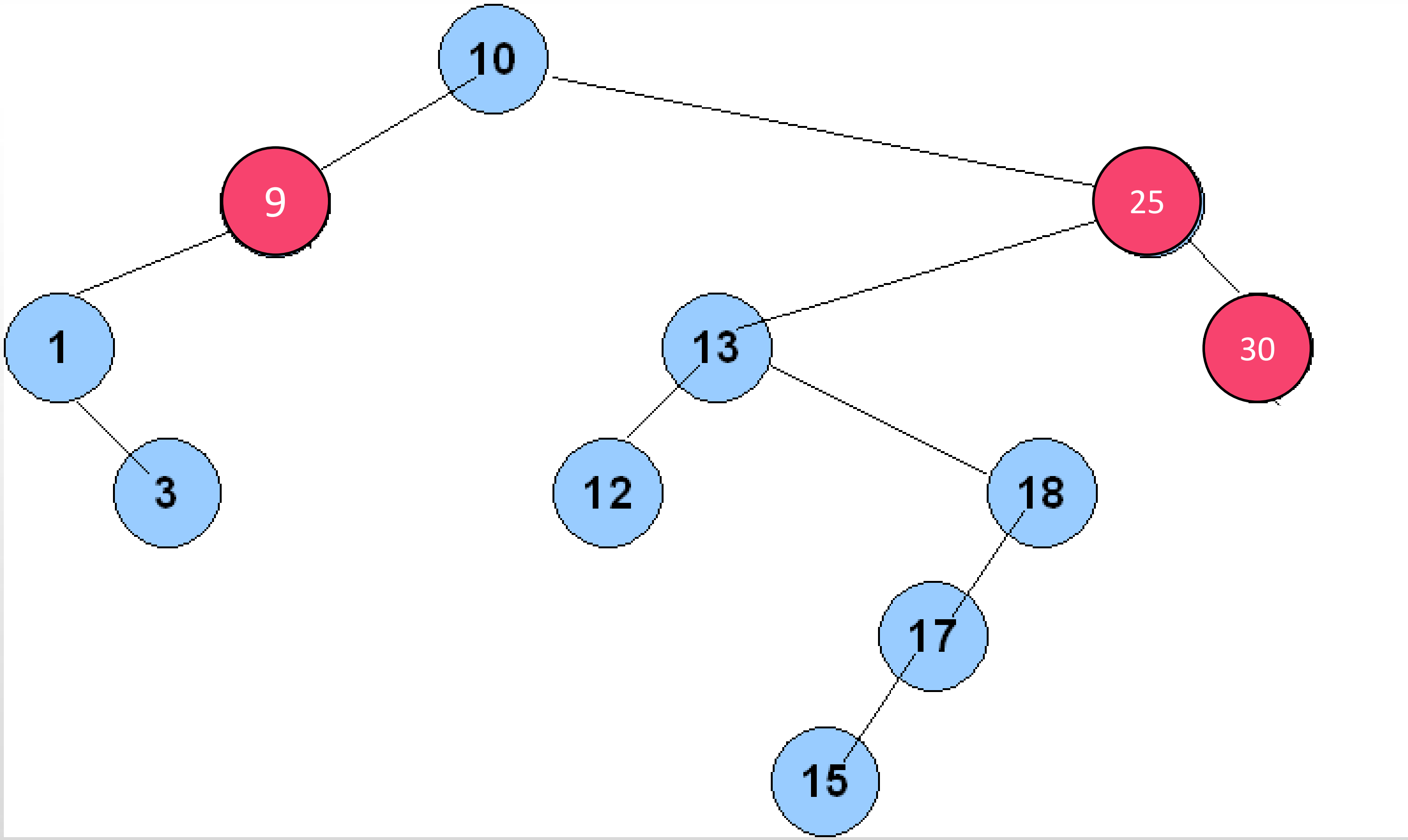
- Delete 5, 21

Deleting non-leaf node with 2 children



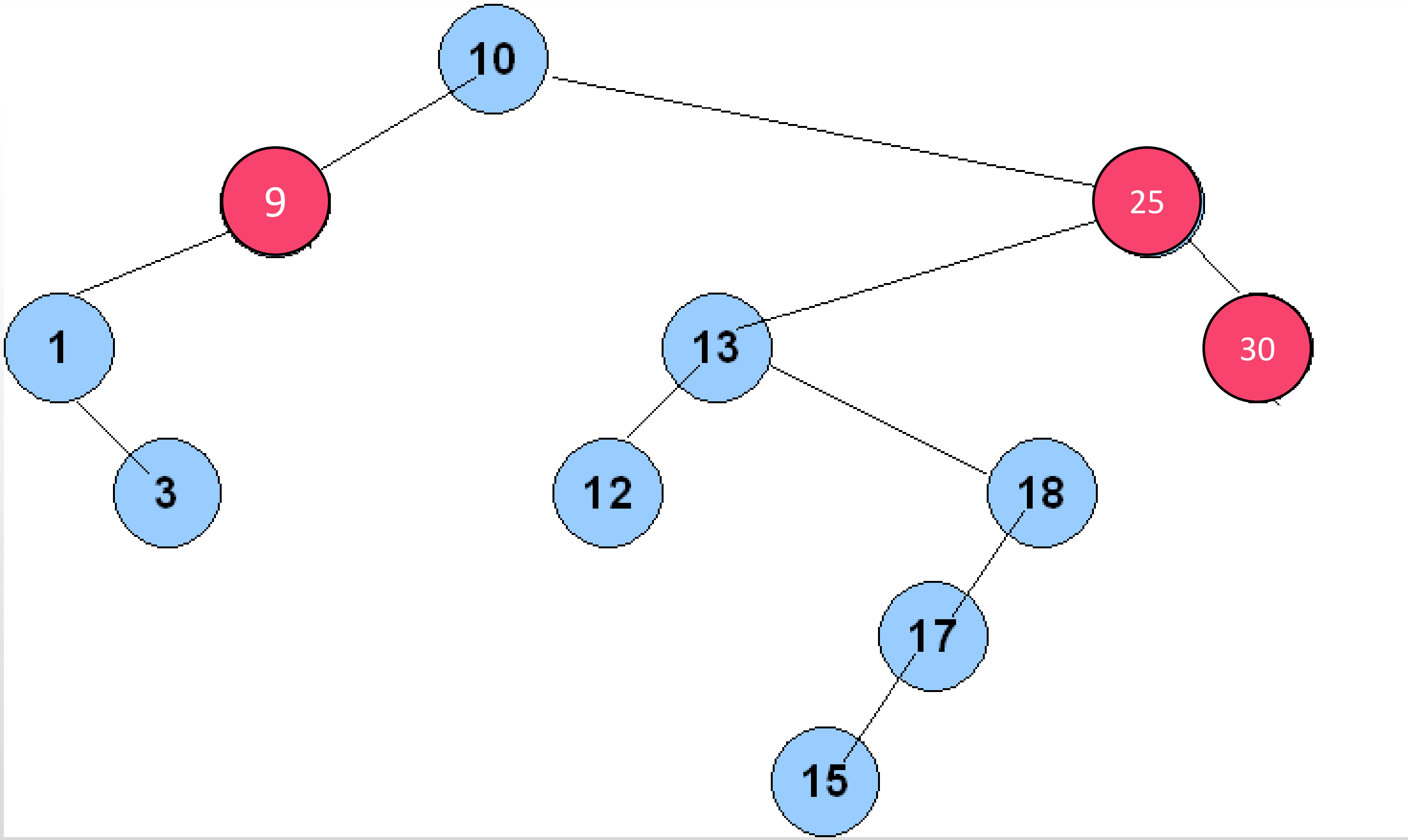
- Delete 5, 21

Deleting non-leaf node with 2 children



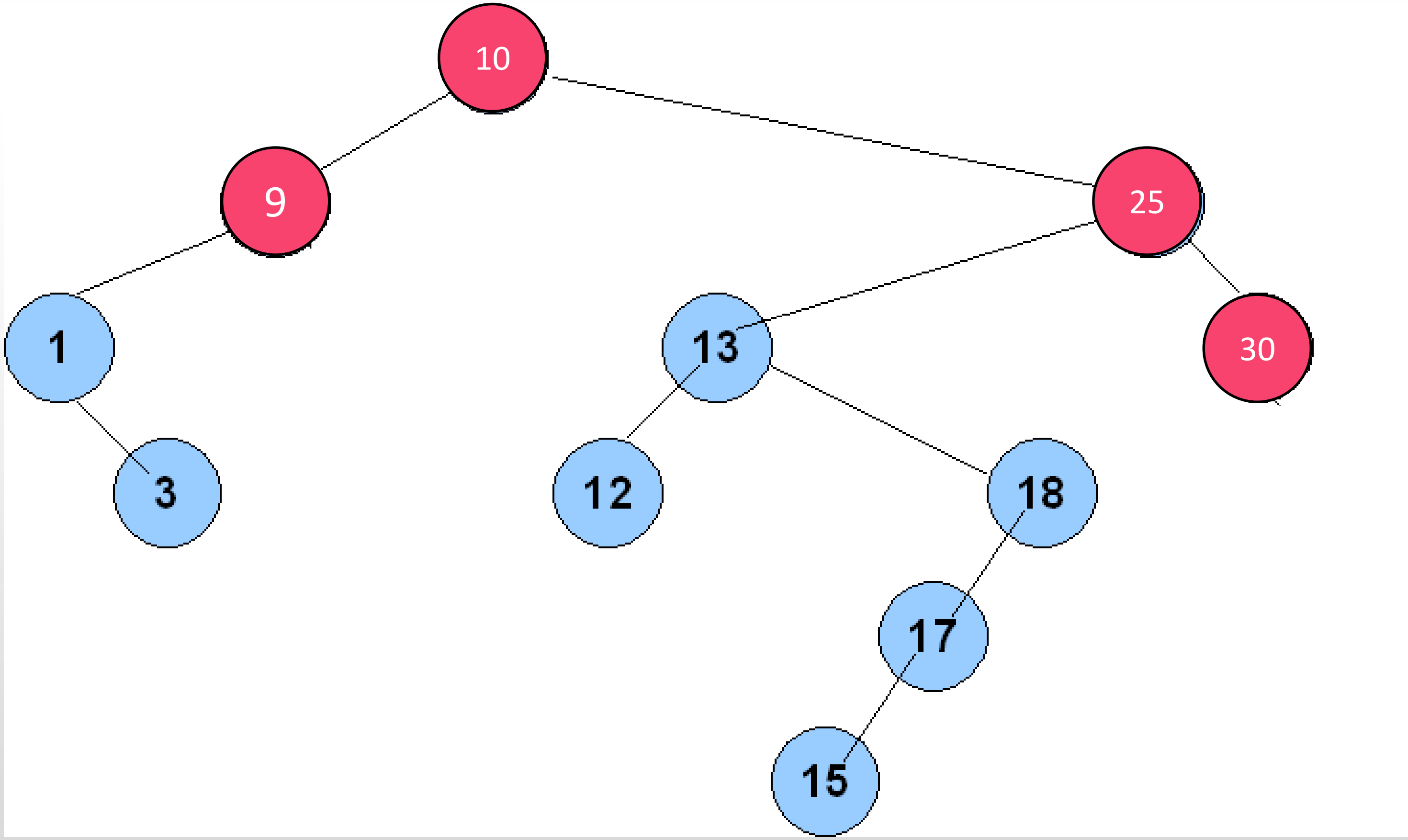
- Delete 5, 21

Deleting non-leaf node with 2 children



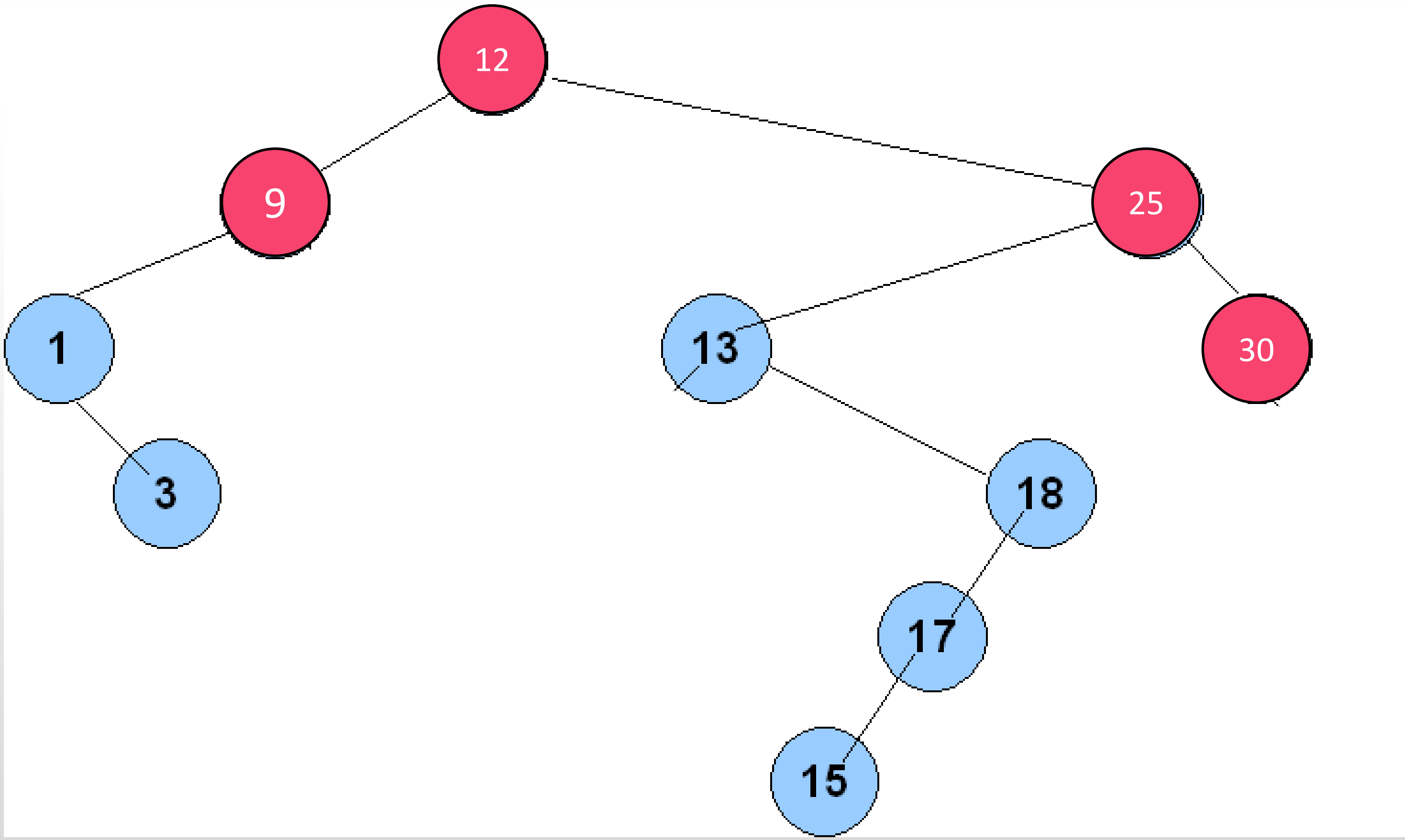
- Delete 5, 21, 10

Deleting non-leaf node with 2 children



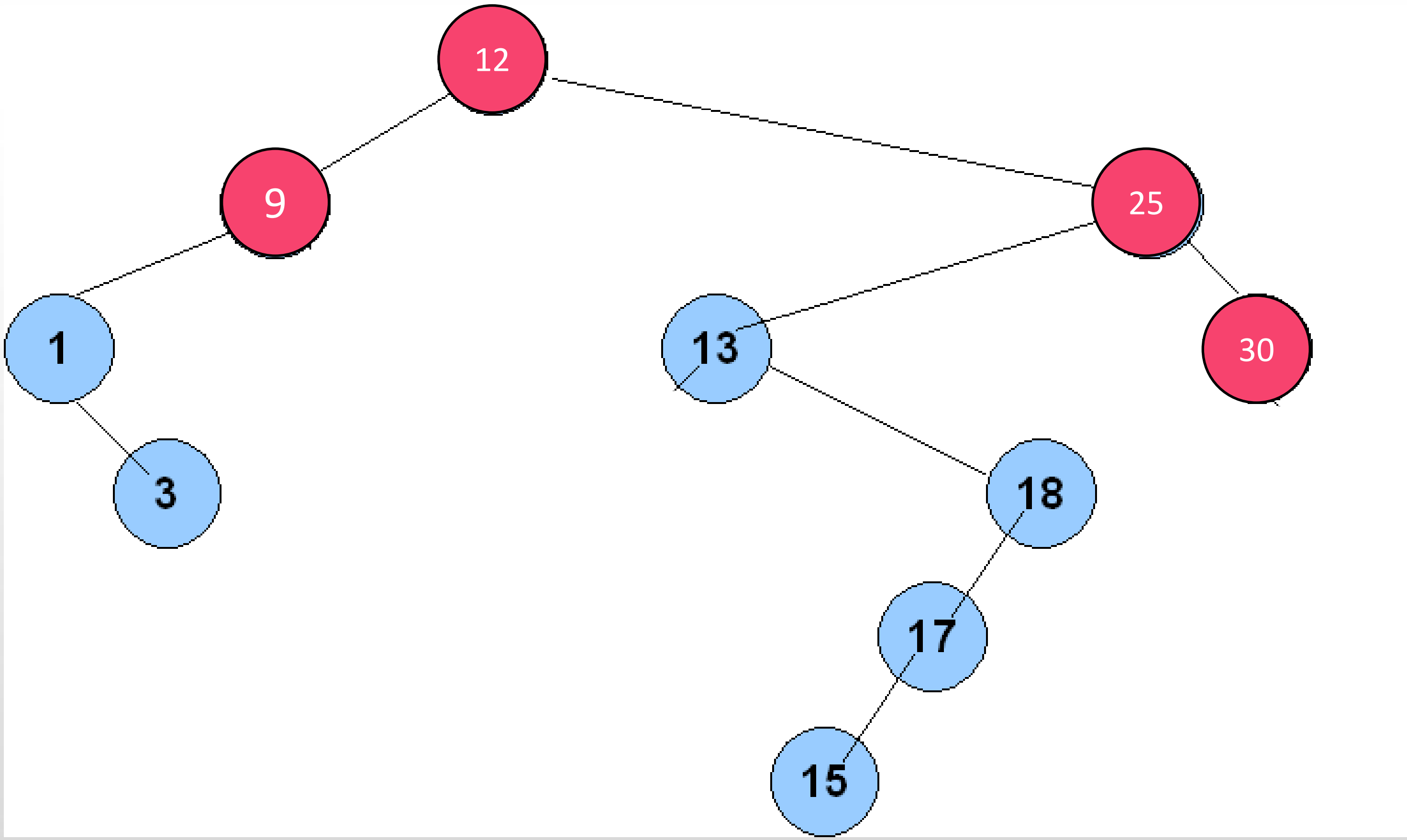
- Delete 5, 21, 10

Deleting non-leaf node with 2 children



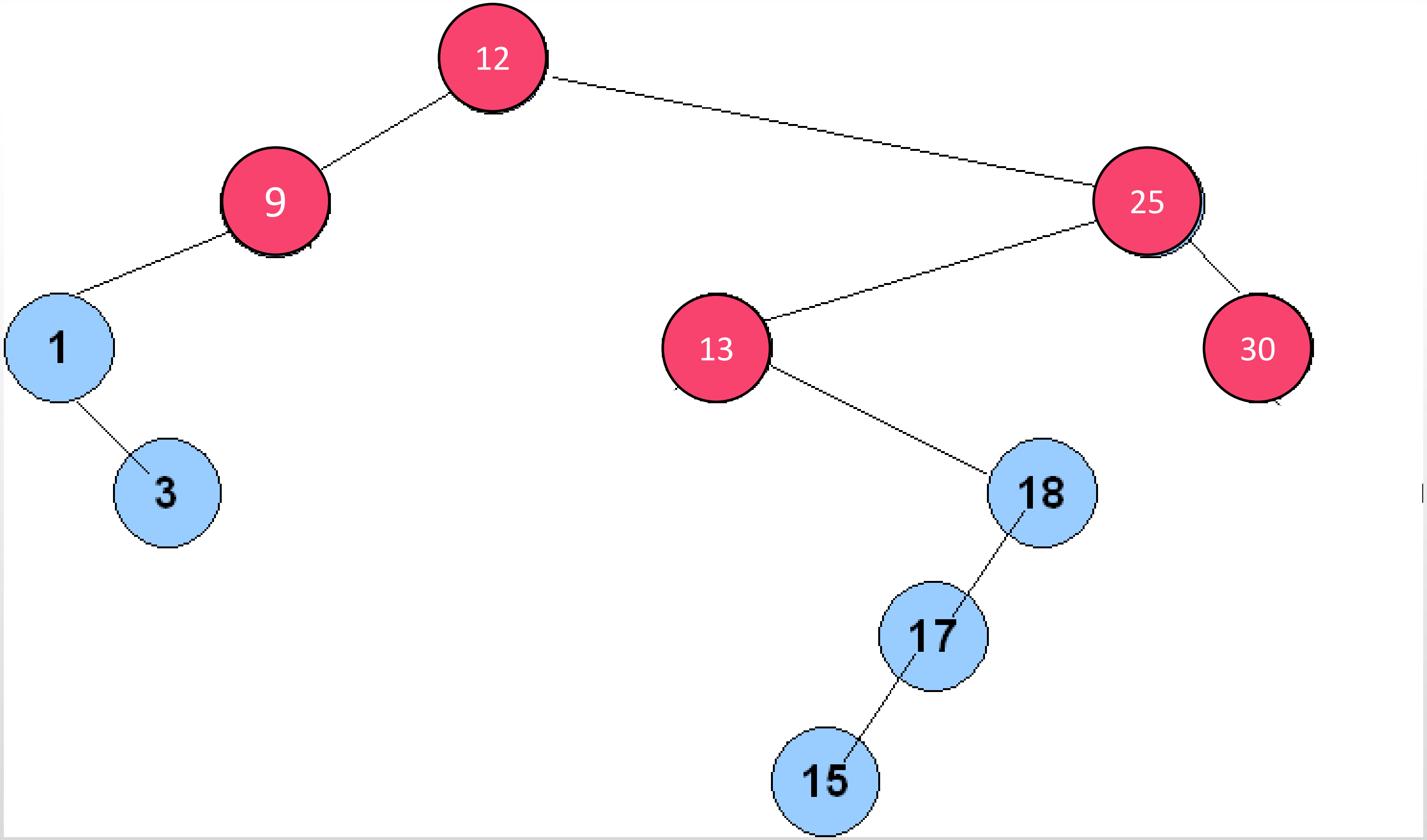
- Delete 5, 21, 10

Deleting non-leaf node with 2 children



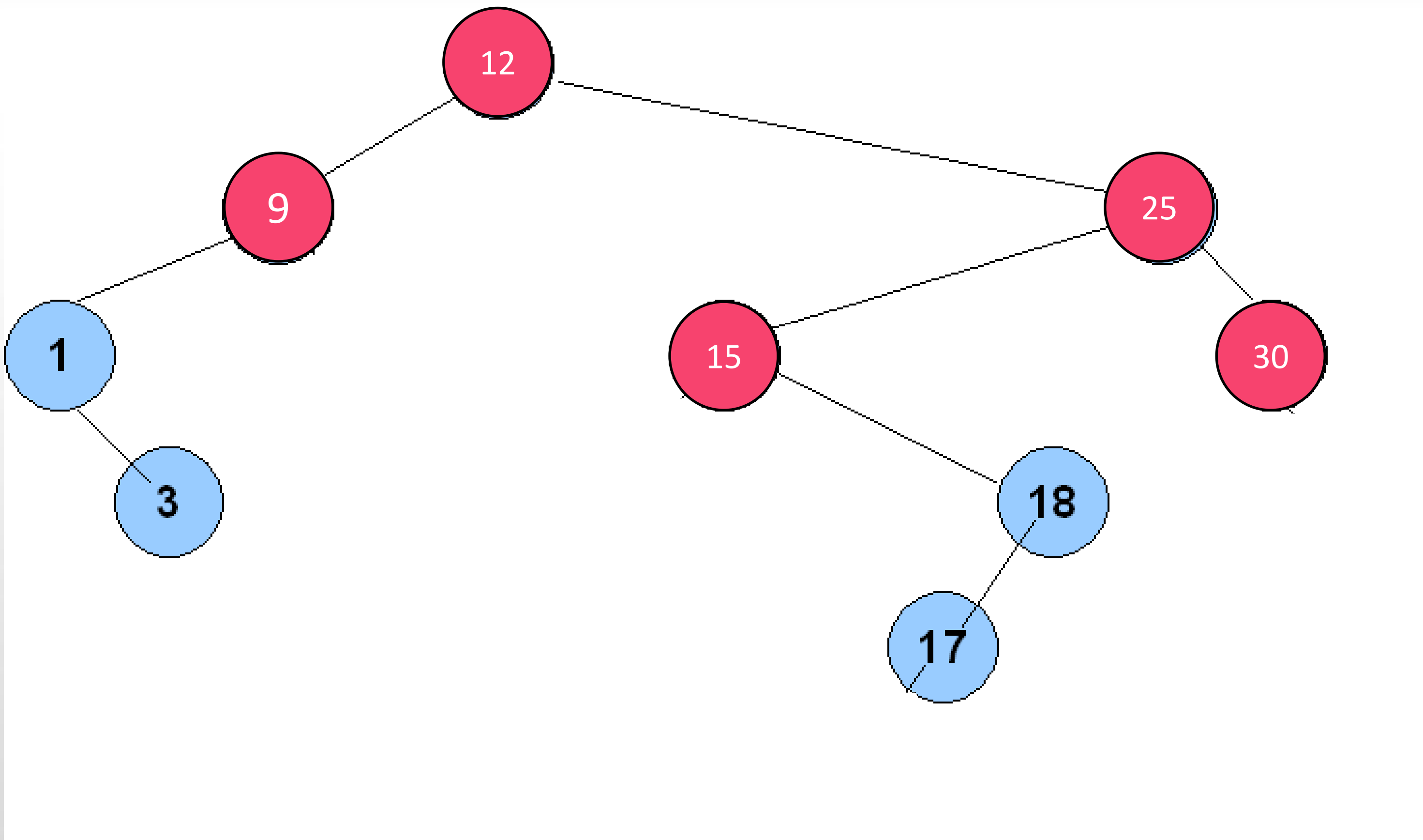
- Delete 5, 21, 10, 13

Deleting non-leaf node with 2 children



- Delete 5, 21, 10, 13

Deleting non-leaf node with 2 children



- Delete 5, 21, 10, 13

Other Operations

- Successor
  - find minimum element in the right subtree
- Predecessor
  - find maximum element in the left subtree

Print BST

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;

int main(){
  BST *root=NULL;

}
```

```
void print_sorted(BST *root){

 if(root!=NULL){
   print_sorted(root->left);
   printf(“%d”, root->value);
   print_sorted(root->right);
 }
}
```

BST - drawback