# Binary Search Tree

Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content.

# *Definitions*

- *Linked List*
  - A *data structure* in which each element is dynamically allocated and in which elements point to each other to define a linear relationship
    - Singly- or doubly-linked
    - Stack, queue, circular list
- Linear access time of linked lists is prohibitive
  - Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(lg N)?

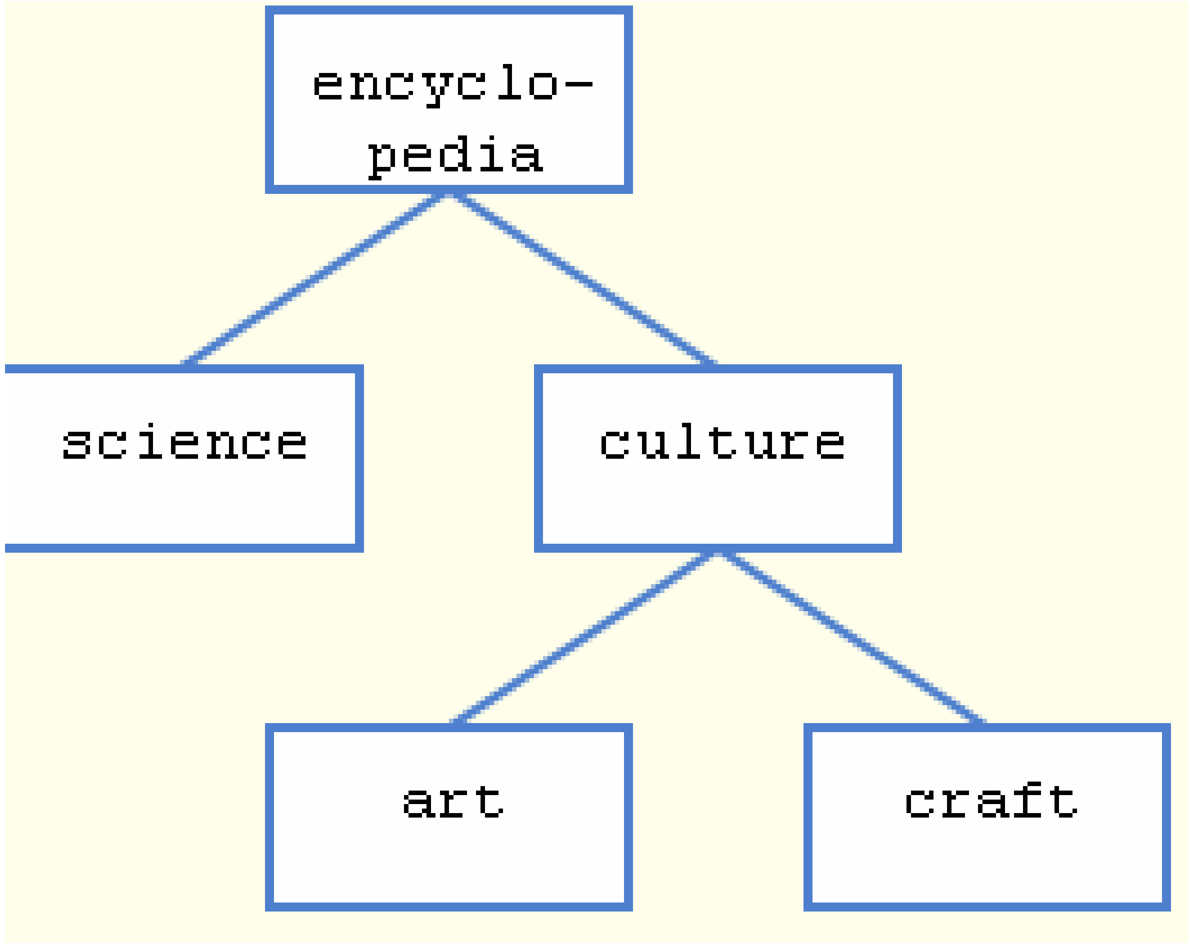
# *Trees*

- *Tree*
  - A *data structure* in which each element is dynamically allocated and in which each element has more than one potential *successor*
    - Defines a *partial order*
- Trees
  - Basic concepts
  - Tree traversal
  - Binary tree
  - Binary search tree and its operations

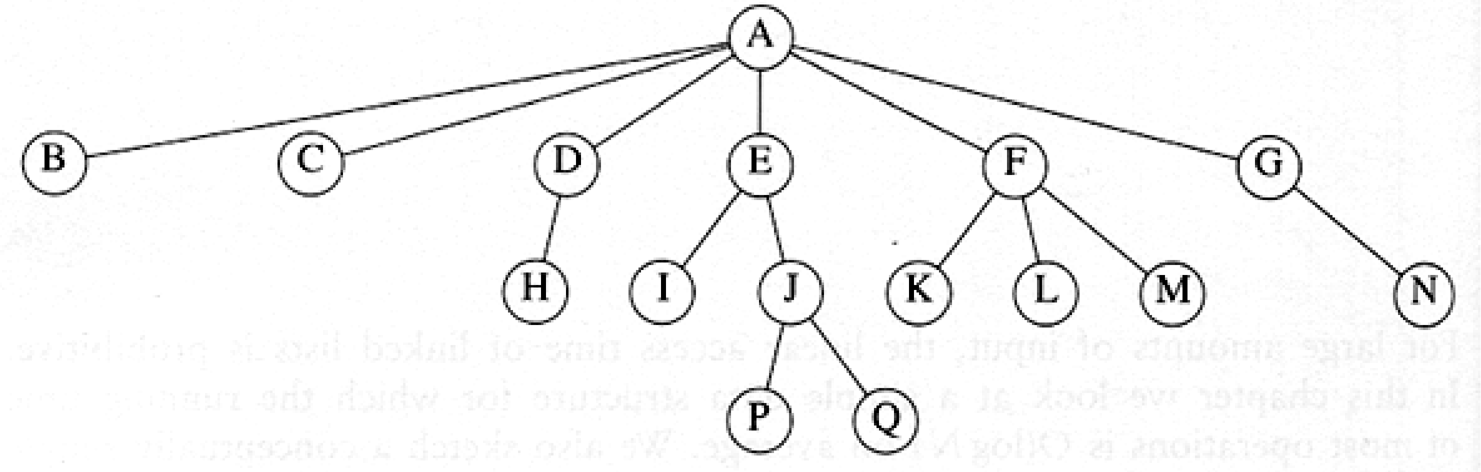
# *Definition of Tree*

- A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - The remaining nodes are partitioned into n>=0 disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree.
  - We call $T_1, \ldots, T_n$ the subtrees of the root.

# *Example*



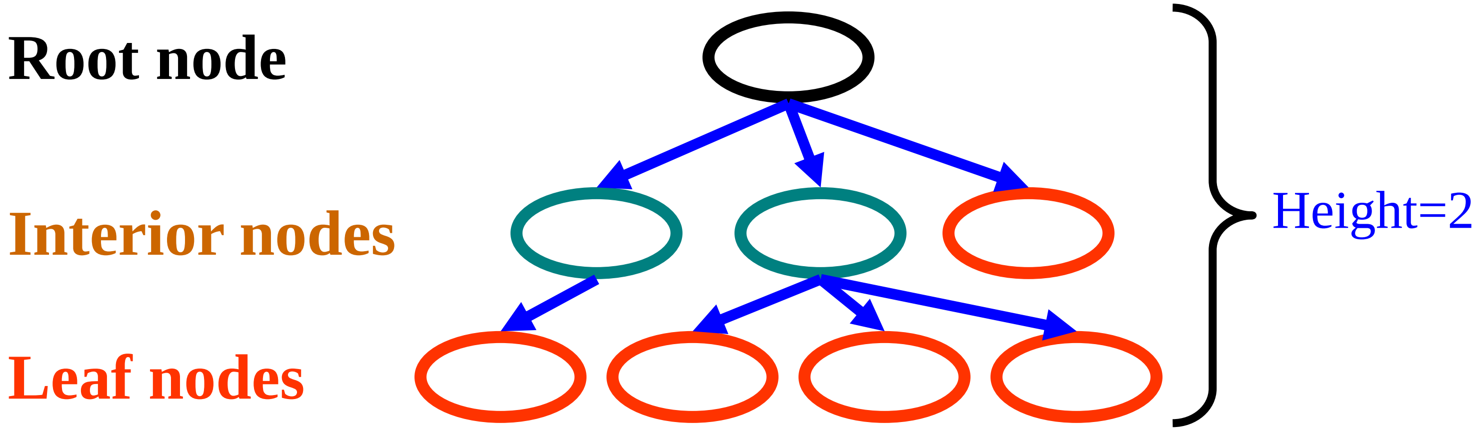
# Some Terminologies




- *Child* and *Parent*
  - Every node except the root has one parent
  - A node can have a zero or more children
- *Leaves*
  - Leaves are nodes with no children
- *Sibling*
  - nodes with same parent

# Terminology

- Root ⇒ no parent
- Leaf ⇒ no child
- Interior ⇒ non-leaf
- Height ⇒ distance from root to leaf

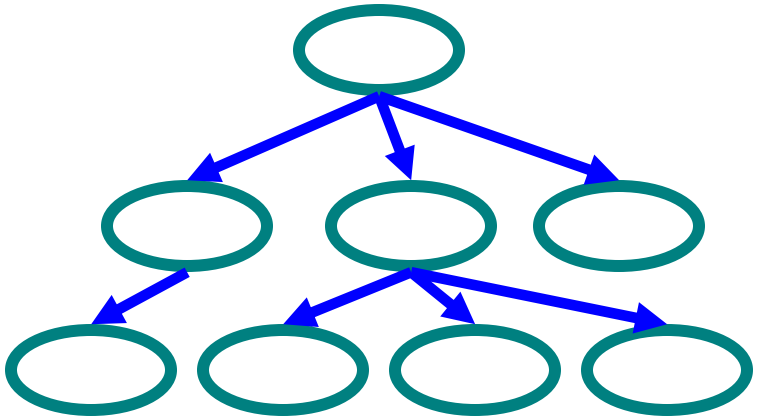


# *More Terminologies*

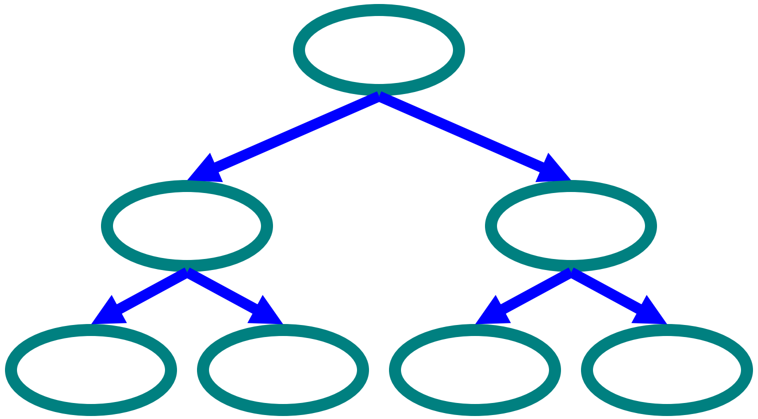
- *Path*
  - A sequence of edges
- *Length of a path*
  - number of edges on the path
- *Depth* of a node
  - length of the unique path from the root to that node
- *Height* of a node
  - length of the longest path from that node to a leaf
  - all leaves are at height 0
- The height of a tree = the height of the root
  = the depth of the deepest leaf
- *Ancestor* and *descendant*
  - If there is a path from n1 to n2
  - n1 is an ancestor of n2, n2 is a descendant of n1

# *Trees Data Structures*

- Tree
  - Nodes
  - Each node can have 0(zero) or more children
  - A node can have at most one parent
- Binary tree
  - Tree with 0–2 children per node



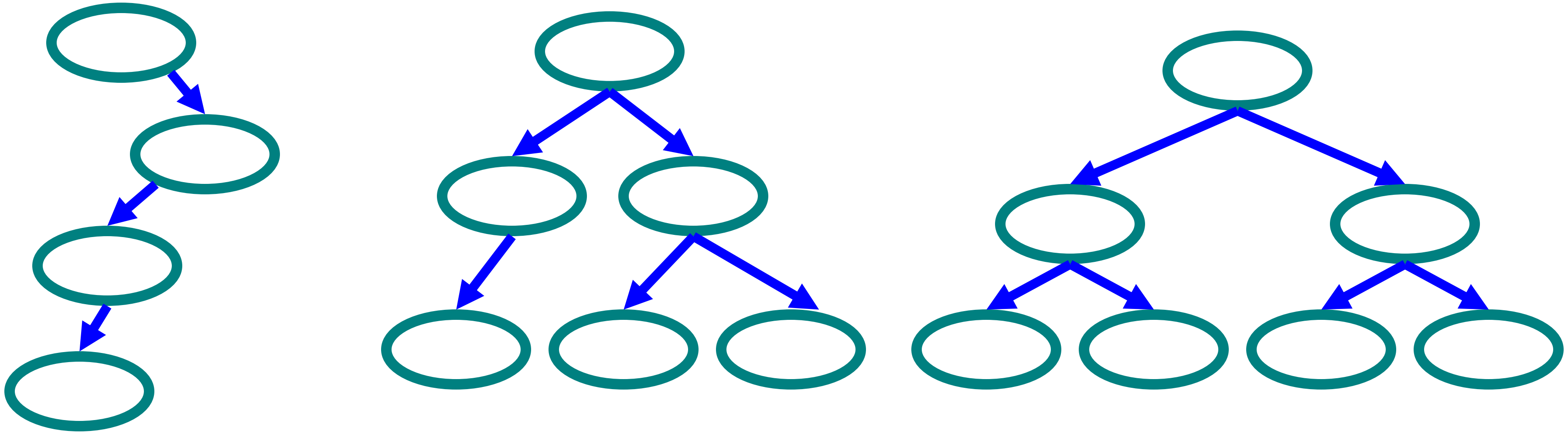
**Tree**



**Binary Tree**

# Types of Binary Trees

- Degenerate – only one child
- Complete – always two children
- Balanced – “mostly” two children
  - more formal definitions exist, above are intuitive ideas



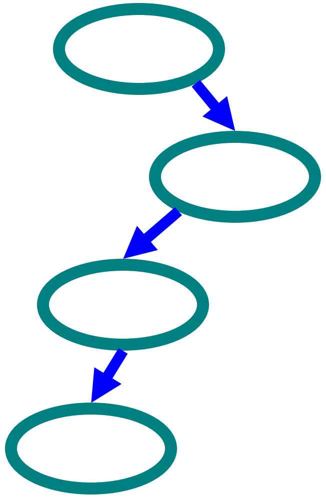
**Degenerate binary tree**

**Balanced binary tree**

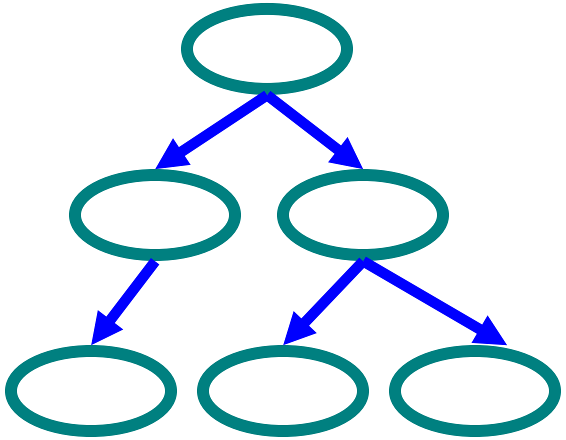
**Complete binary tree**

# *Binary Trees Properties*

- Degenerate
  - Height = O(n) for n nodes
  - Similar to linked list
- Balanced
  - Height = O( log(n) ) for n nodes
  - Useful for searches



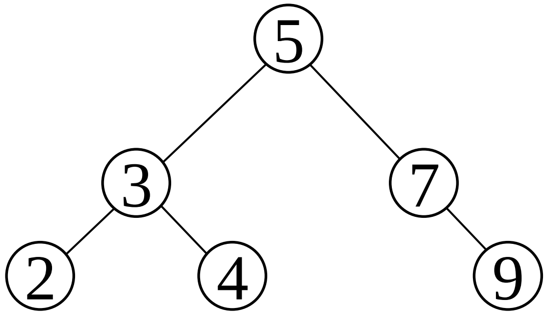
**Degenerate binary tree**



**Balanced binary tree**

# *Binary Search Tree Property*

- Binary search tree property:
  - If y is in left subtree of x, then key [y] < key [x]



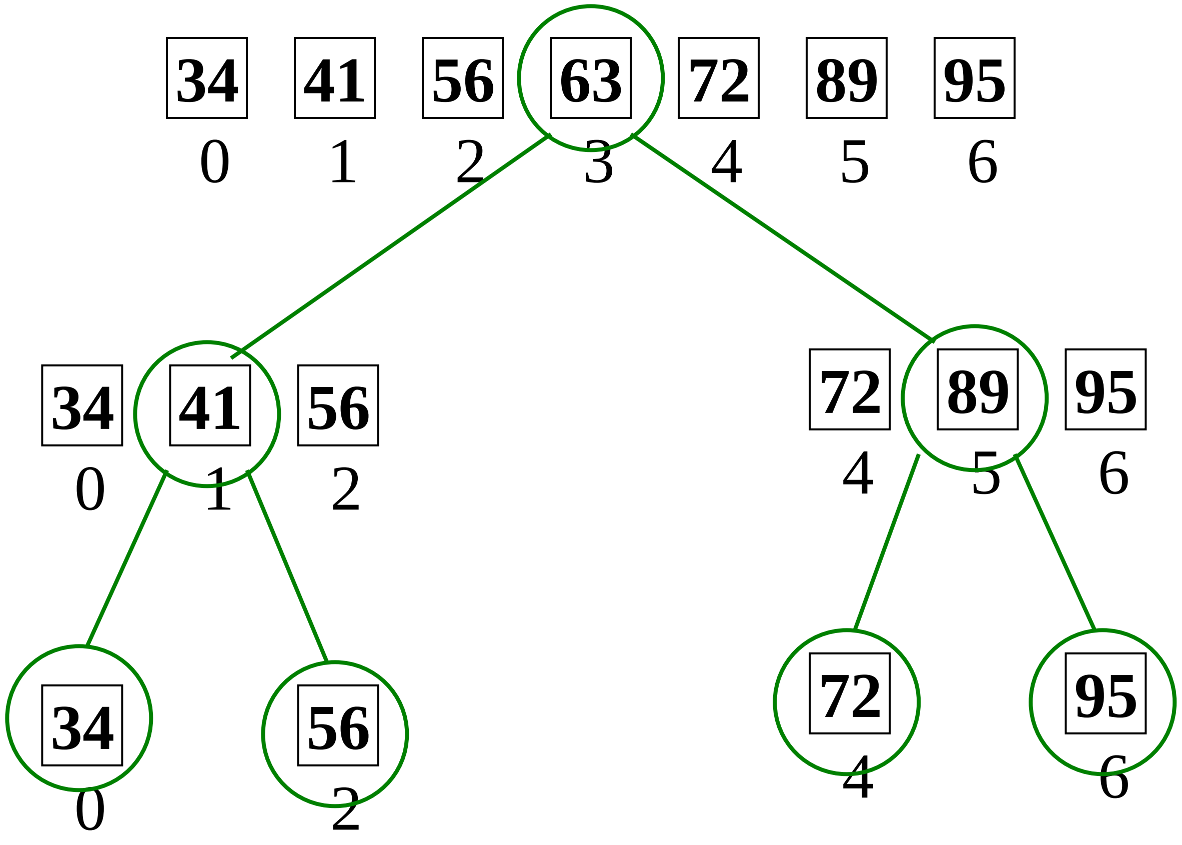

  - If y is in right subtree of x, then key [y] > key [x]

# *Binary Search Trees*

- Support many dynamic set operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Running time of basic operations on binary search trees
  - On average: $\Theta(\lg n)$
    - The expected height of the tree is $\lg n$
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of $n$ nodes
    - O(height) in general

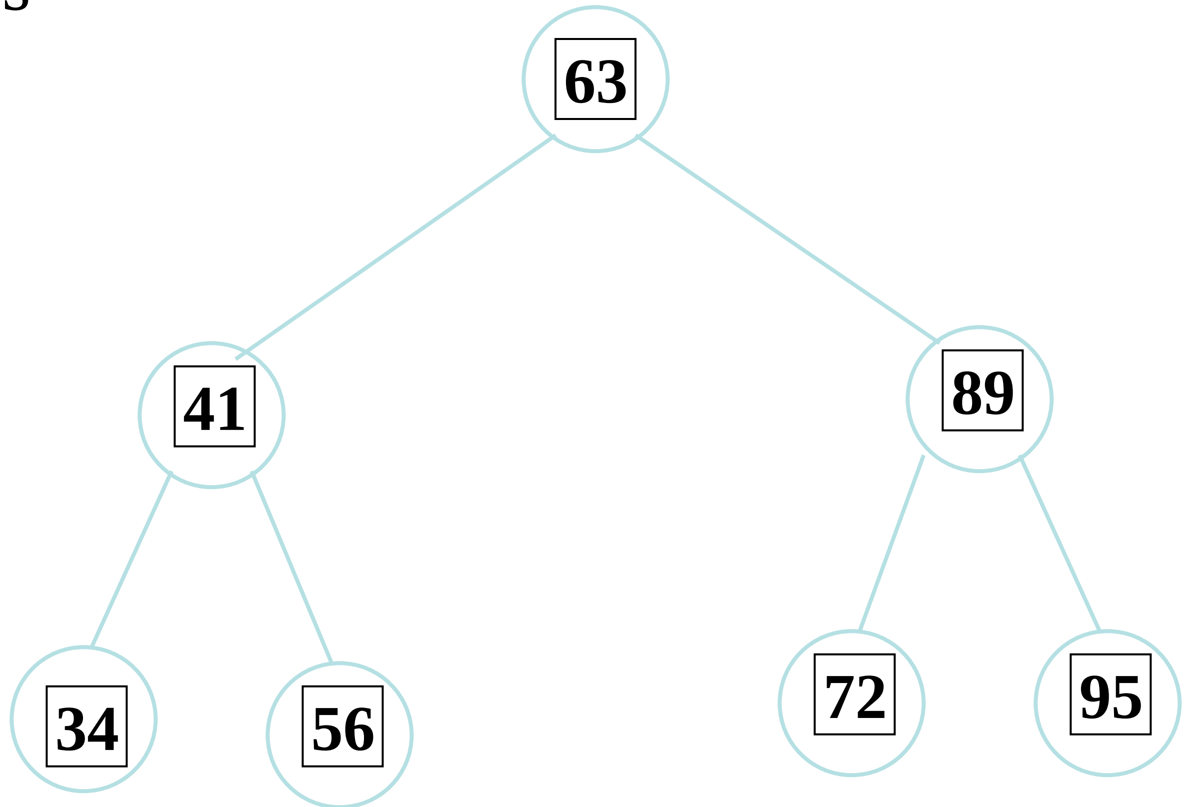
# *Binary Search Algorithm*

Binary Search algorithm of an array of *sorted* items reduces the search space by one half after each comparison

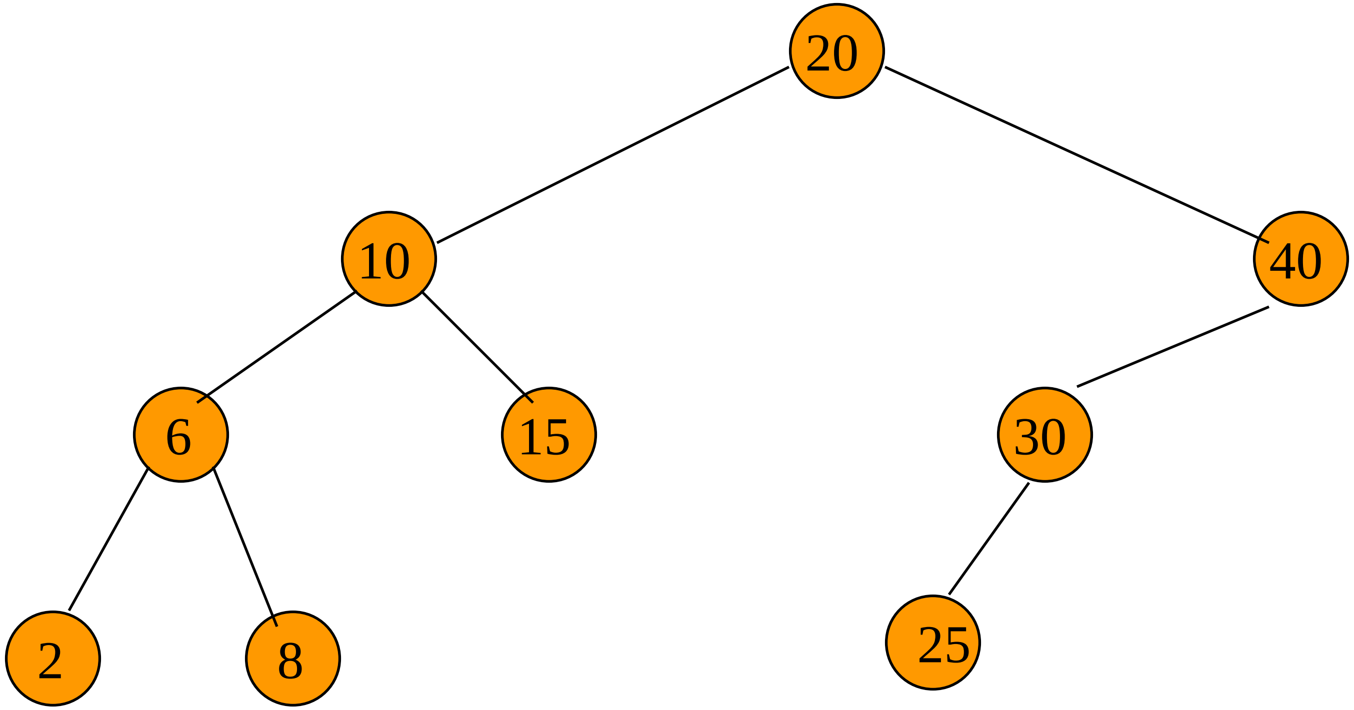


# *Organization Rule for BST*

- The values in all nodes in the left subtree of a node are less than the node value
- The values in all nodes in the right subtree of a node are greater than the node values



# Example Binary Search Tree



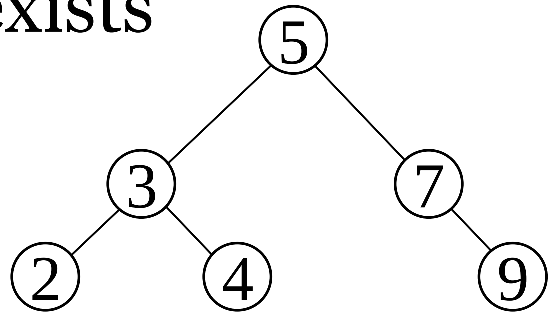

Only keys are shown.

# *Binary Tree*

```
struct node {
        int  data;
        struct node *rchild;
        struct node *lchild;
  } ;
typedef struct node* ptrnode;
ptrnode root;
```

# Searching for a Key

- Given a pointer to the root of a tree and a key k:
  - Return a pointer to a node with key k if one exists
  - Otherwise return NIL




- Idea
  - Starting at the root: trace down a path by comparing k with the key of the current node:
    - If the keys are equal: we have found the key
    - If k < key[x] search in the left subtree of x
    - If k > key[x] search in the right subtree of x

# Search in BST - Pseudocode

```
if the tree is empty
        return NULL
else if  the item in the node equals the “target”
        return the node
else if  the item in the node is greater than the “target”
        return the result of searching the left subtree
else if  the item in the node is smaller than the target
        return the result of searching the right subtree
```

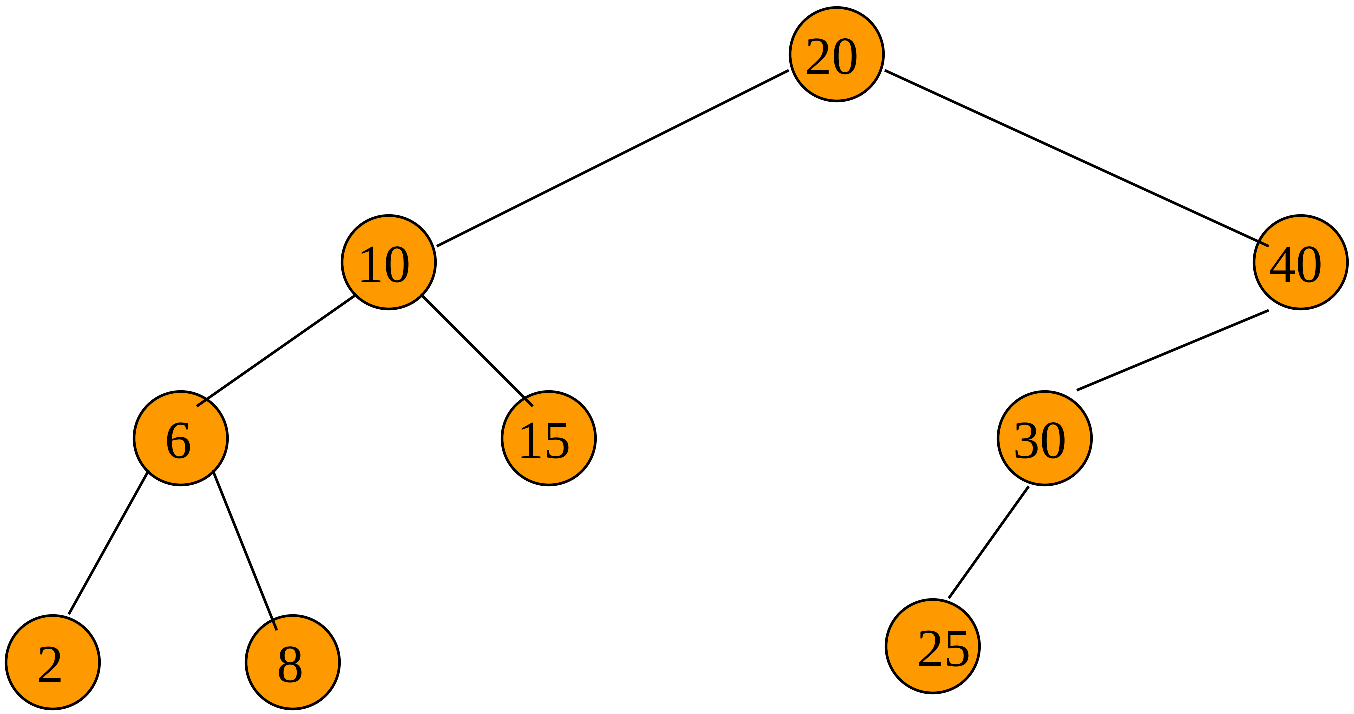
# *Iterative Search of Binary Tree*

```
struct node *search ( ptrnode root, int key) {
    while (root != NULL) {
        if (root->data == key)        // Found it
            return root;
        else if (root->data > key)    // In left subtree
            root = root->lchild;
        else                          // In right subtree
            root = root->rchild;
    }
    return NULL;
}
ptrnode  head = search( root, 5);
```

# *Recursive Search in a BST*

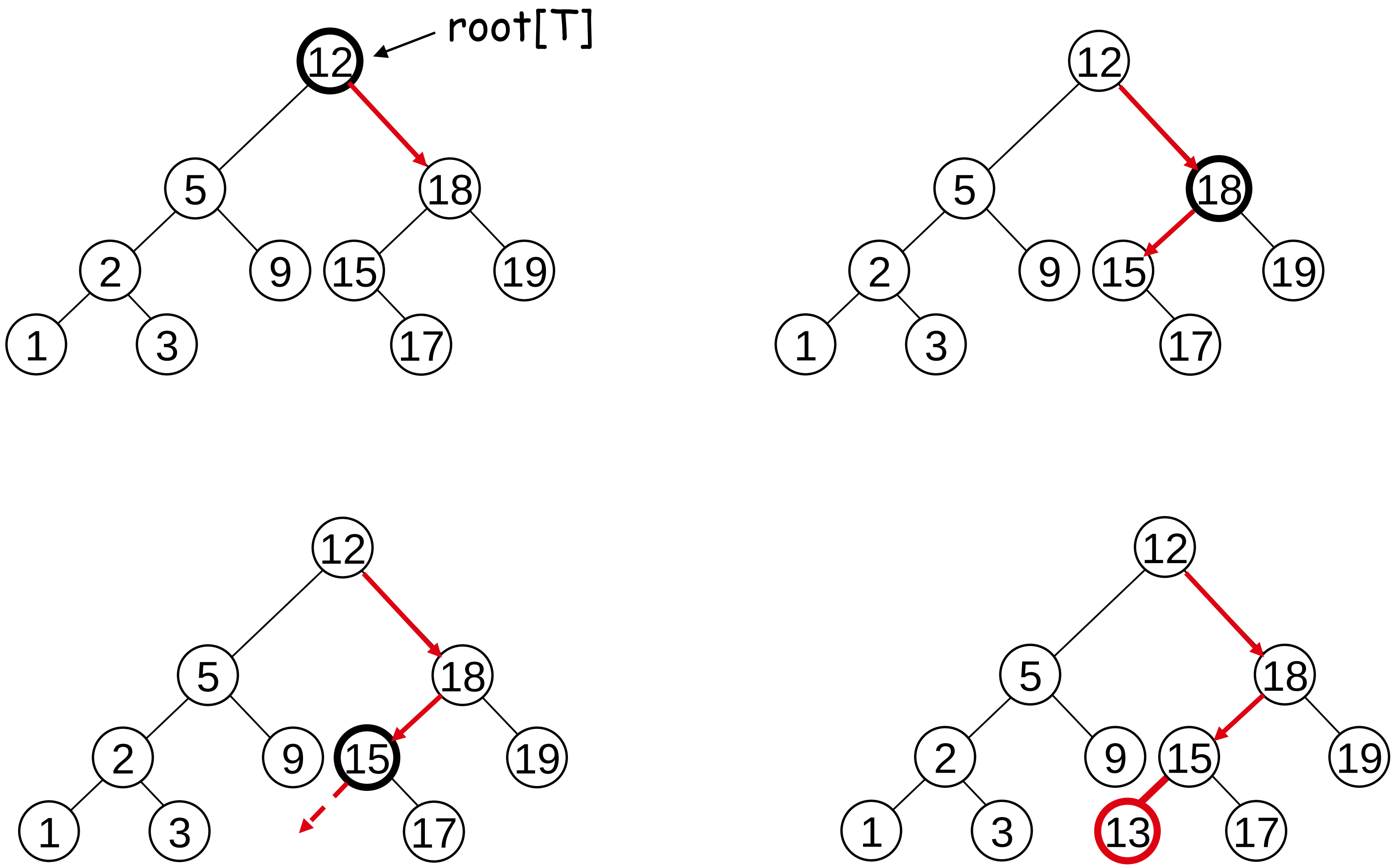
```
ptrnode search(ptrnode root, int key)
{
/* return a pointer to the node that  contains key.
   If there is no such   node, return NULL */

  if (root==NULL) return NULL;
  else if (key ==root->data) return root;
  else if (key < root->data)
       return search(root-> lchild, key);
  else return search(root-> rchild, key);
}
```

# The Operation search( )




Complexity is O(height) = O(n) in the worst case, where n is number of nodes/elements.

# Example: INSERT (13)




Complexity of insert( ) is O(height).

# Insertion in BST - Pseudocode

```
if  tree is empty
    create a root node with the new key
else
    compare key with the root node
    if key =  node key
        Duplicate key found
    else if  key >  node key
        compare key with the root of the right subtree:
        if  subtree is empty create a leaf node
        else Insert key  in right subtree
    else  key <  node key
        compare key with the left subtree:
        if the subtree is empty create a leaf node
        else Insert key to the left subtree
```

# BST – Implementation – Insertion

```
ptrnode Insert(ptrnode root, int key){
  if( root == NULL ){
    /* Create and return a one-node tree */
    newnode = (ptrnode)malloc( sizeof(node));
    if( newnode == NULL )
      Errormessage( "Out of space!!!" );
    else {
      newnode -> data = key;
      newnode->lchild = newnode->rchild = NULL;
    }
   return newnode;
  }
  else if( key < root -> data )
    root->lchild = Insert(root->lchild, key);
  else if( key > node->data)
    root->rchild = Insert(root->rchild,key);
  /* Else key is in the tree already; do nothing */
  return root;  /* Do not forget this line!! */
}
```

# BST Shapes

- The order of supplying the data determines where it is placed in the BST, which determines the shape of the BST
- Create BSTs from the same set of data presented each time in a different order:

a) 17 4 14 19 15 7 9 3 16 10

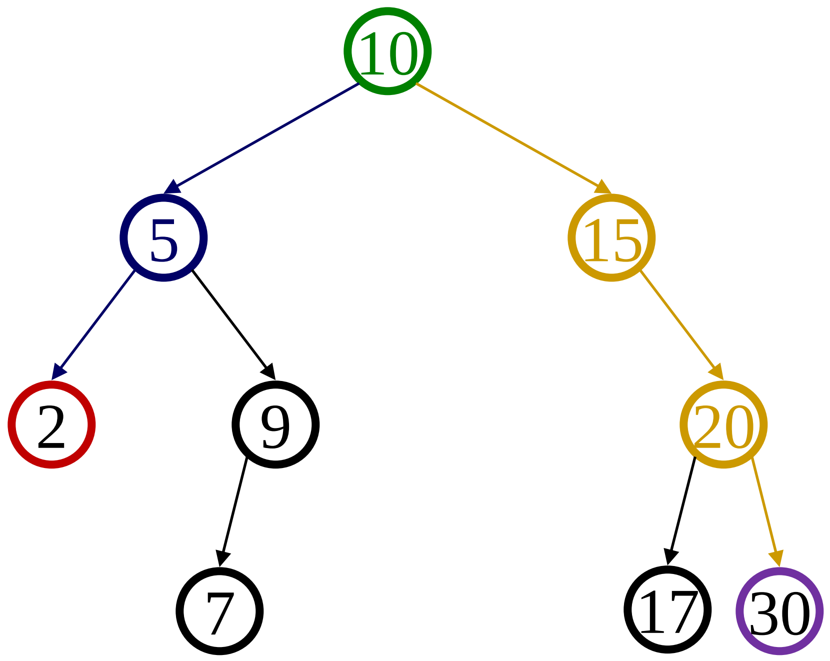
b) 9 10 17 4 3 7 14 16 15 19

c) 19 17 16 15 14 10 9 7 4 3 can you **guess** this shape?

# *FindMin, FindMax*

- Find Minimum

- Find Maximum

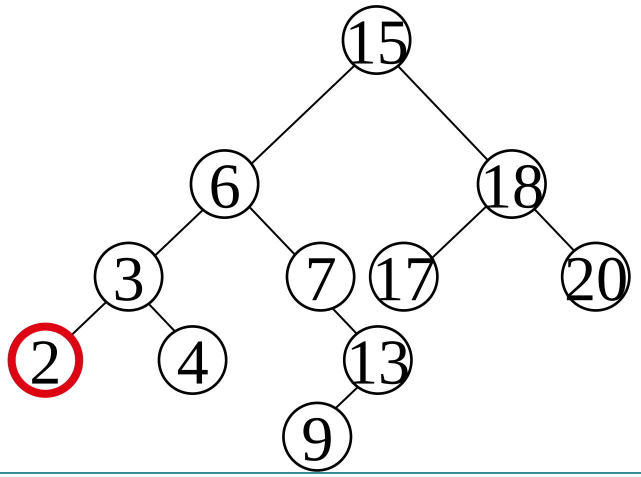


# *Finding Min & Max*

- The binary-search-tree property guarantees that:
  - » The minimum is located at the left-most node.

<u>Tree-Minimum(*node*)</u>

```
ptrnode  minimum(ptrnode root)
 while (root->lchild != NULL)
          root=root->lchild;
 return root;
```



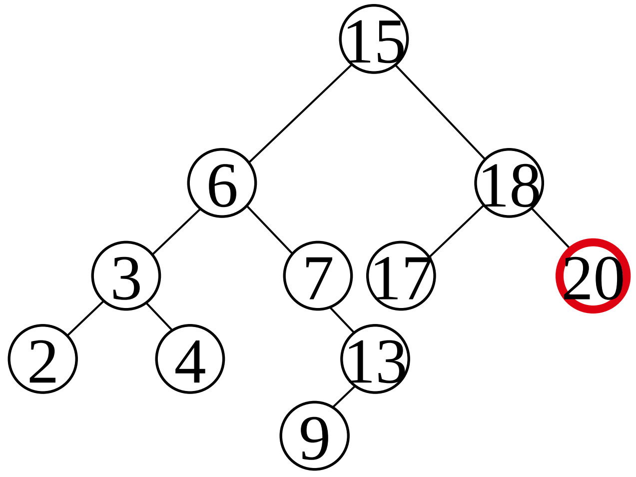

➢ Complexity: O(height)

# *Finding Min & Max*

- The binary-search-tree property guarantees that:
  - »The maximum is located at the right-most node.

Tree-Maximum(node)

```
ptrnode  maximum(ptrnode root)
while (root->rchild != NULL)
            root=root->rchild;
 return root;
```

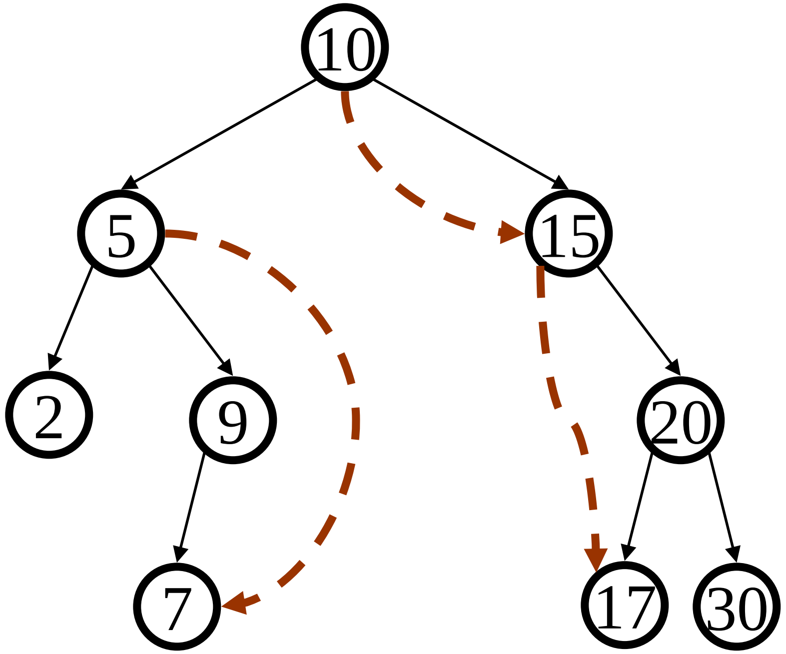



➢ Complexity: O(height)

# *Successor Node*

- **Next larger node in node's subtree**
  - Leftmost node in the right subtree

```
ptrnode succ(ptrnode node){
   if (node->rchild == NULL)
      return NULL;
   else
      return minimum(node->rchild);
}
```

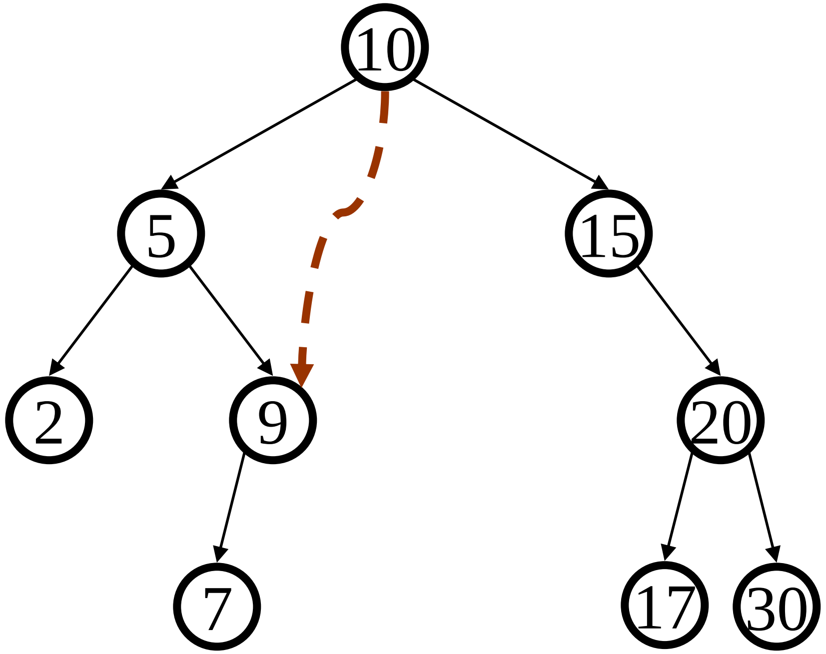



*How many children can the successor of a node have?*

# *Predecessor Node*

- Next smaller node in node's subtree
  - Rightmost node in the left subtree

```
ptrnode pred(ptrnode node) {
  if (node->lchild == NULL)
      return NULL;
  else
    return maximum(node->lchild);
}
```



# The Operation Delete()

Three cases:

- Element is in a leaf.
- Element is in a degree 1 node.
- Element is in a degree 2 node.

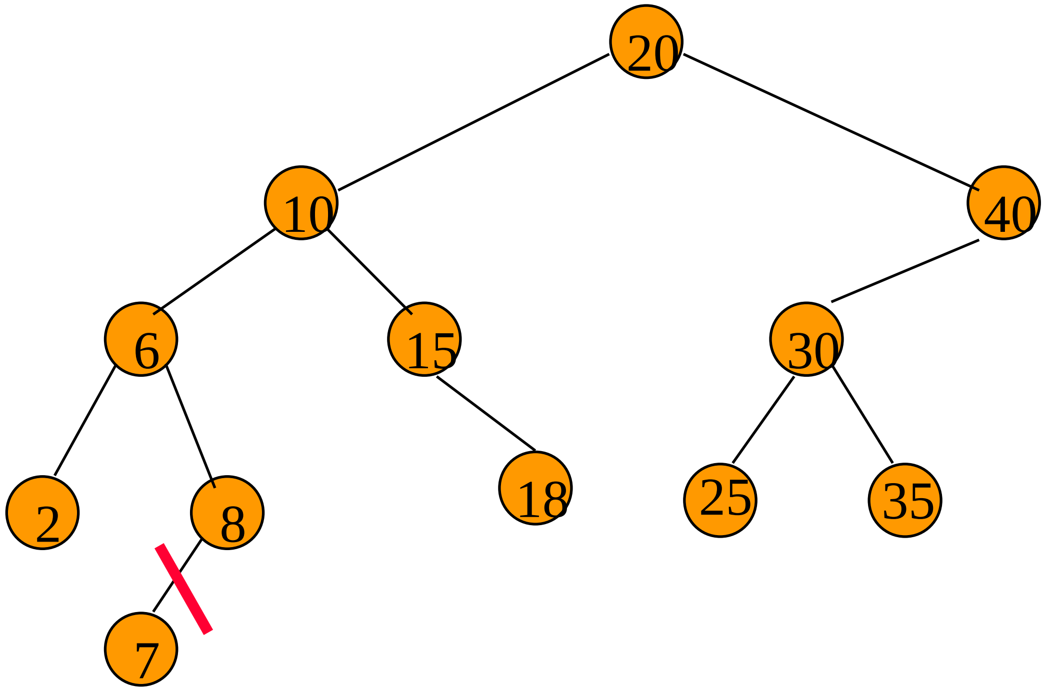
# Delete a Leaf node (key:7)

Idea:

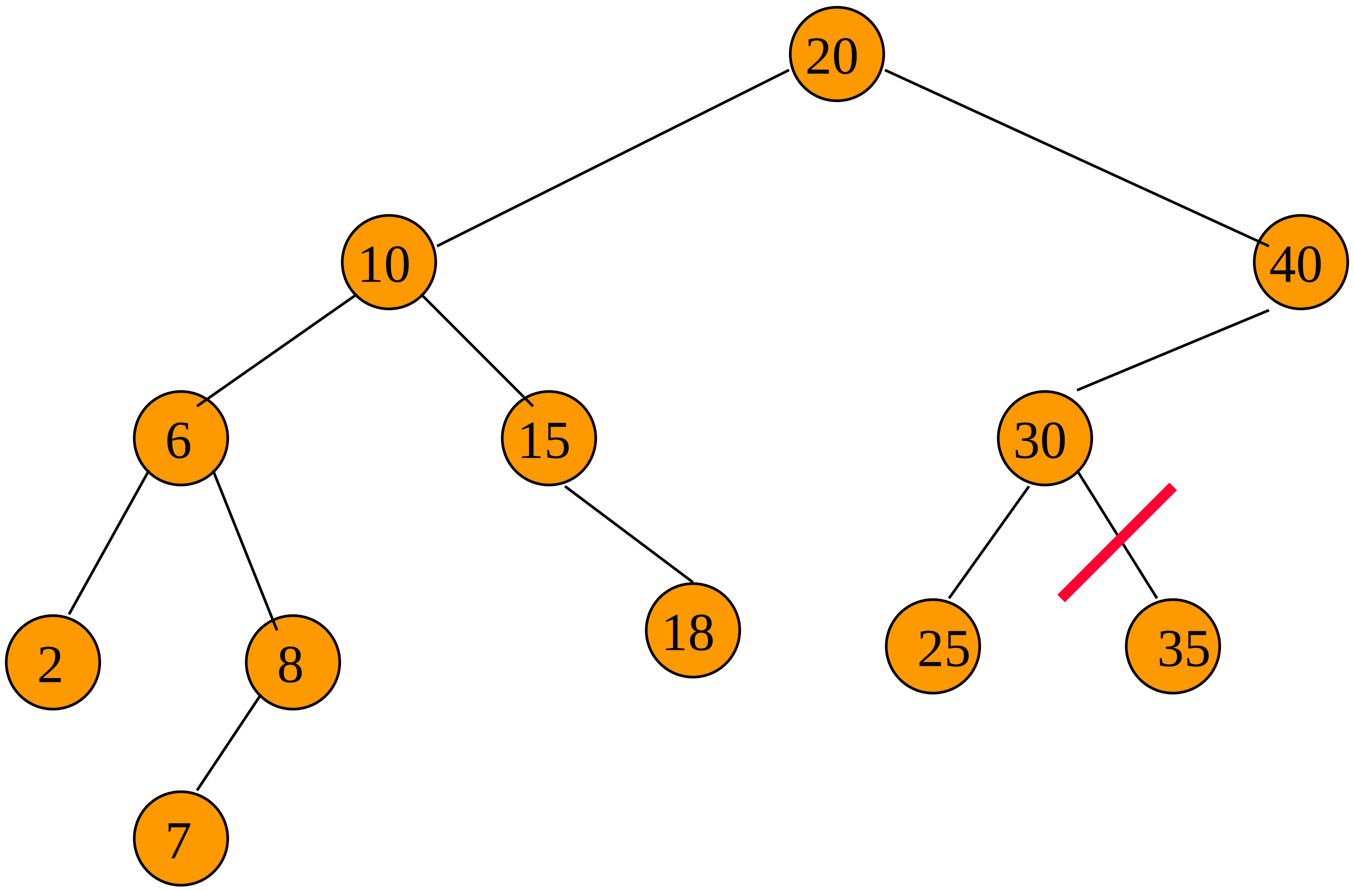
**Case 1:** node **z** has no children

Delete **z** by making the parent of **z** point to NULL

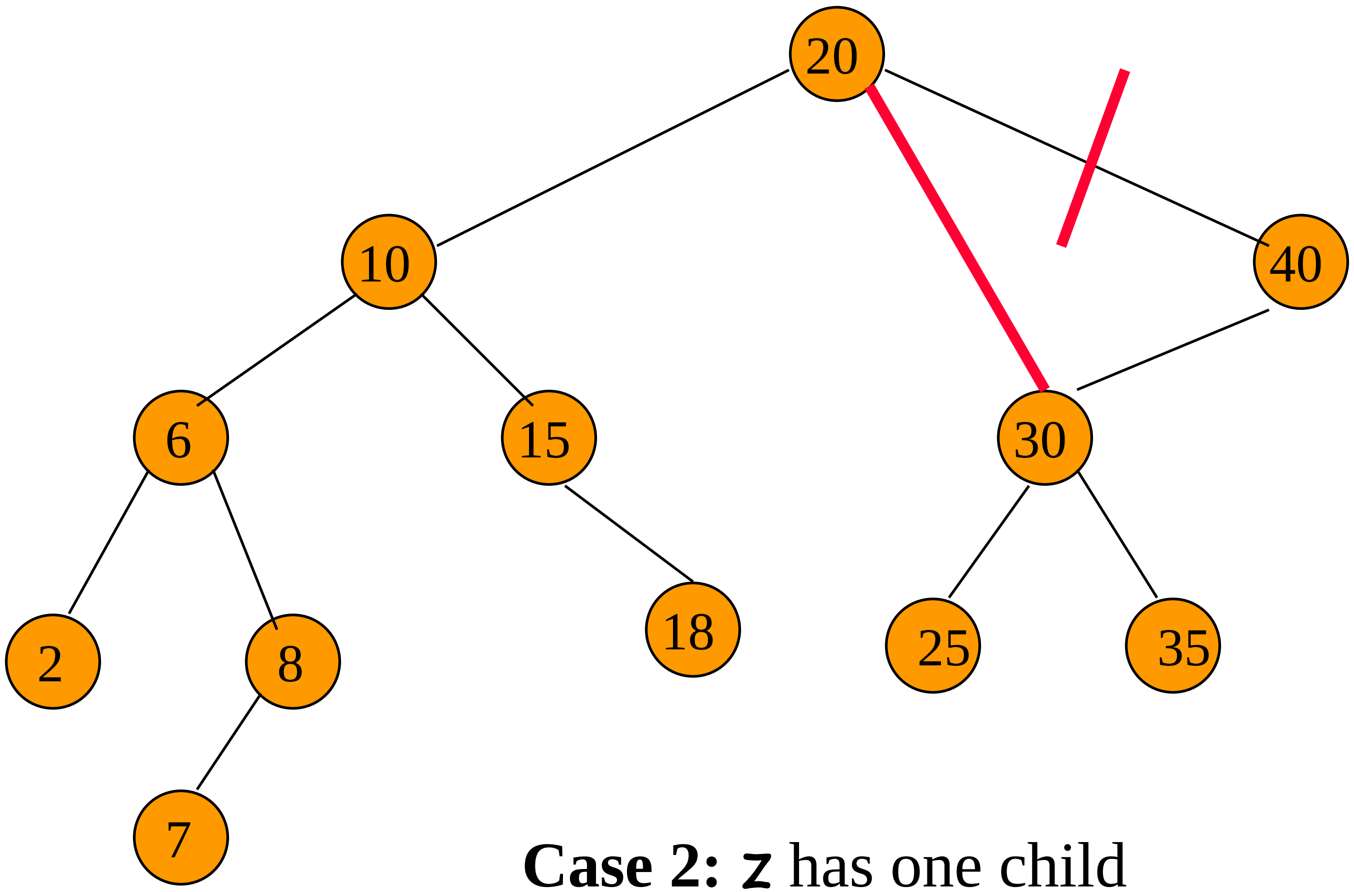
```
ptrnode toDelete = current;
current=current->lchild;(NULL)
Or
current=current->rchild;(NULL)
free(toDelete);
```



# Delete a Leaf node (key:35)



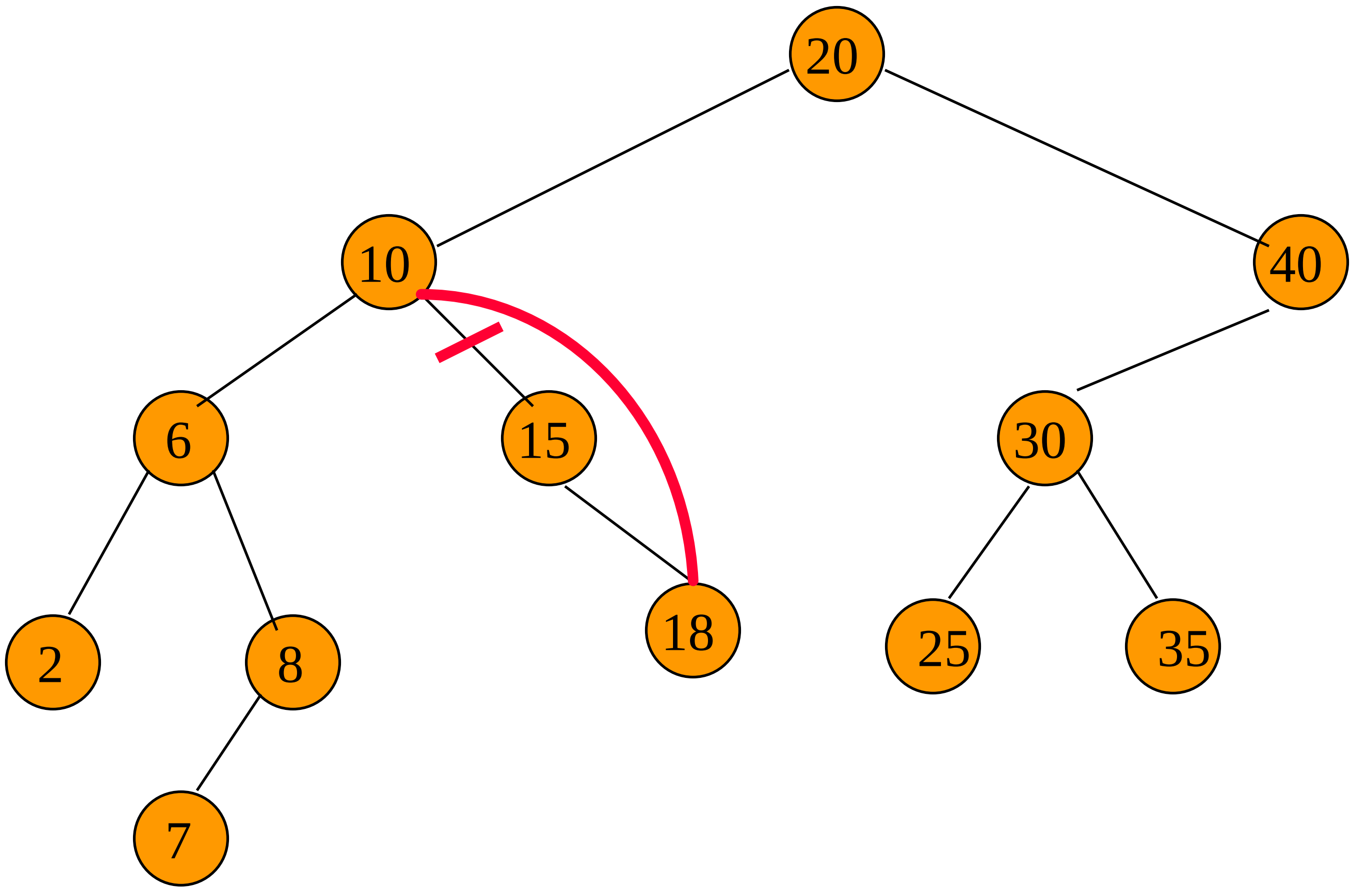
# Delete a Degree 1 Node (Key: 40)



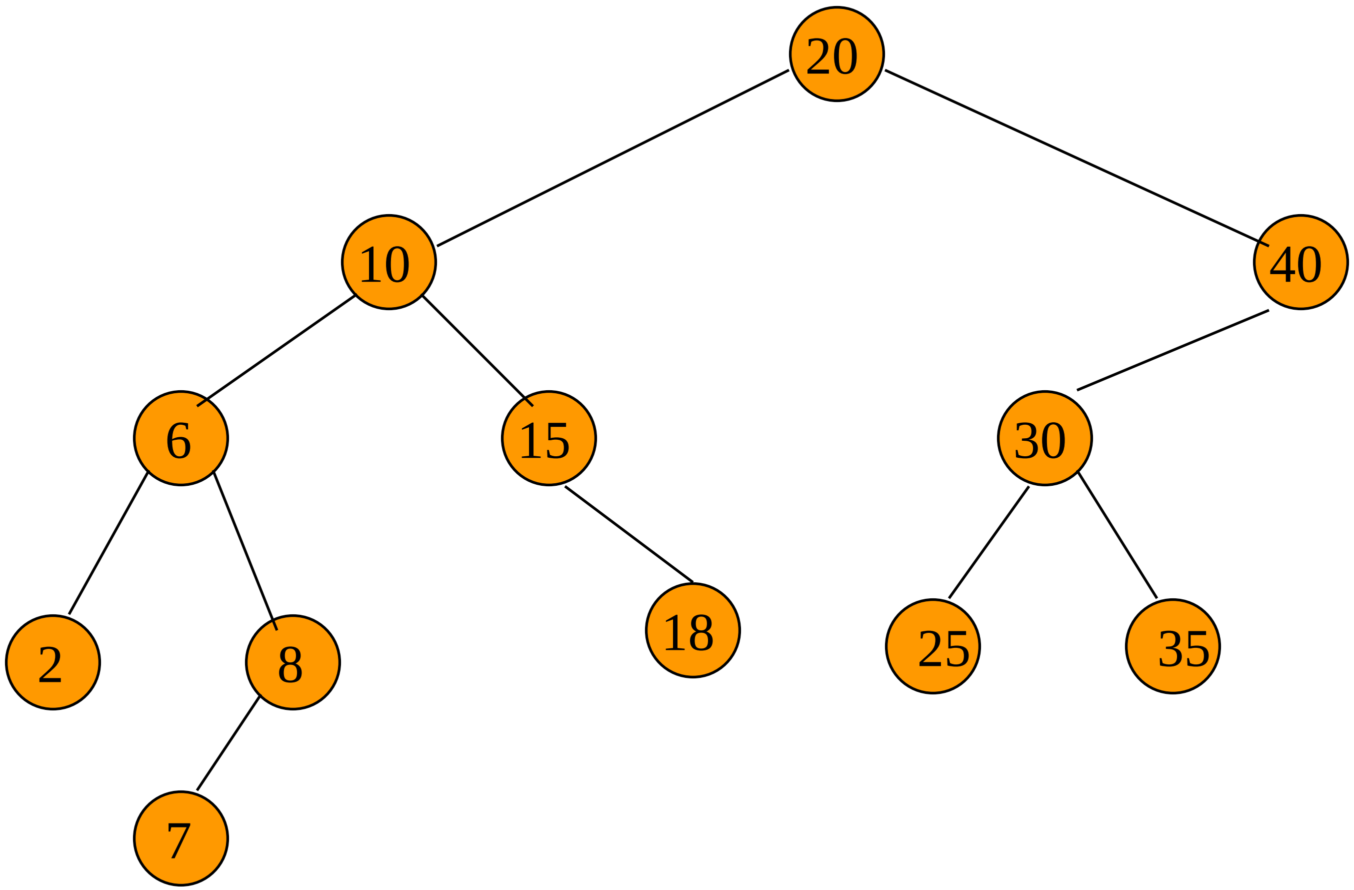

**Case 2:** ***z*** has one child

Delete ***z*** by making the parent of ***z*** point to ***z***'s child, instead of to ***z***

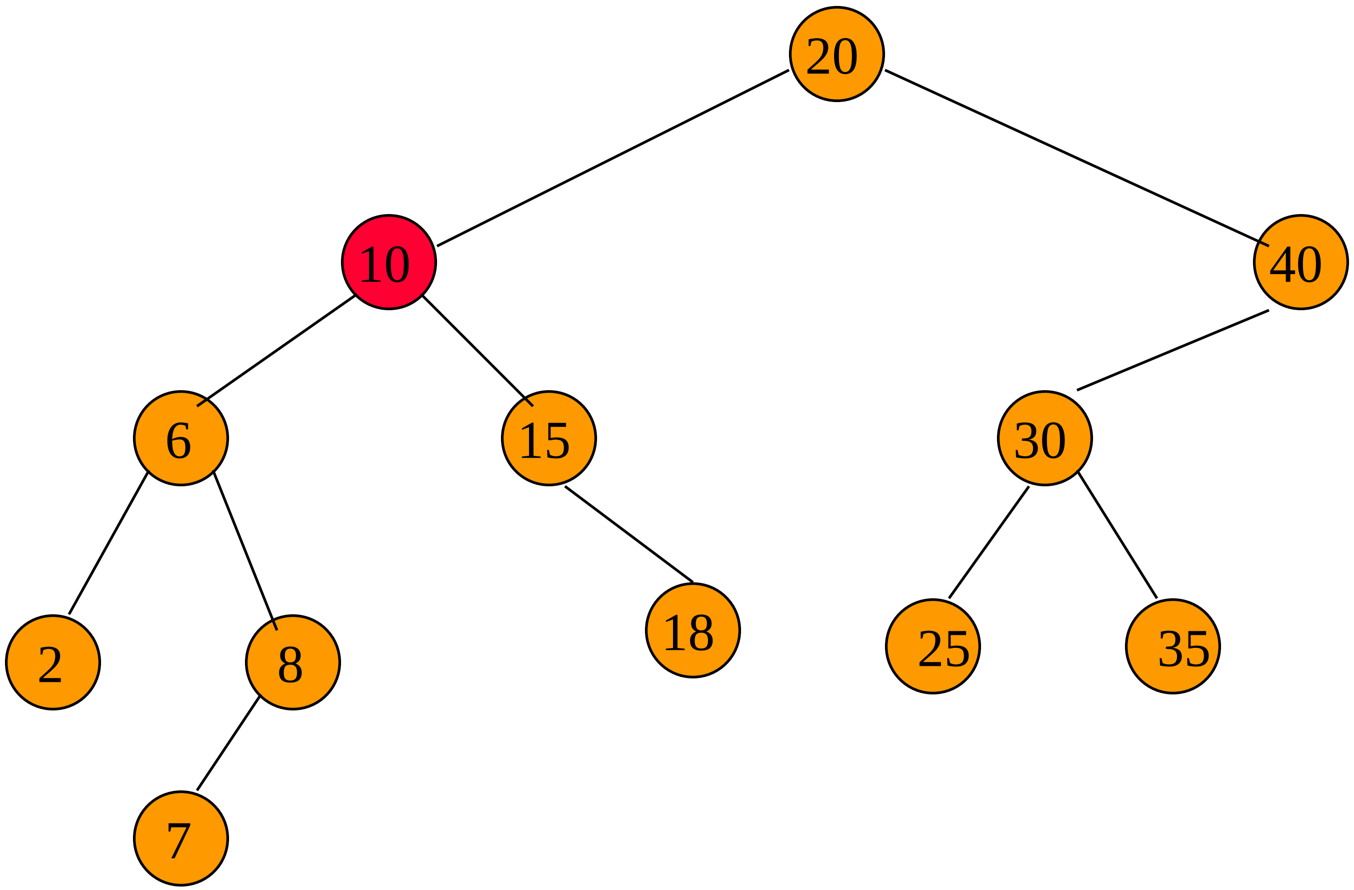
# Delete a Degree 1 Node (key:15)



# Delete a Degree 2 Node (key:10)

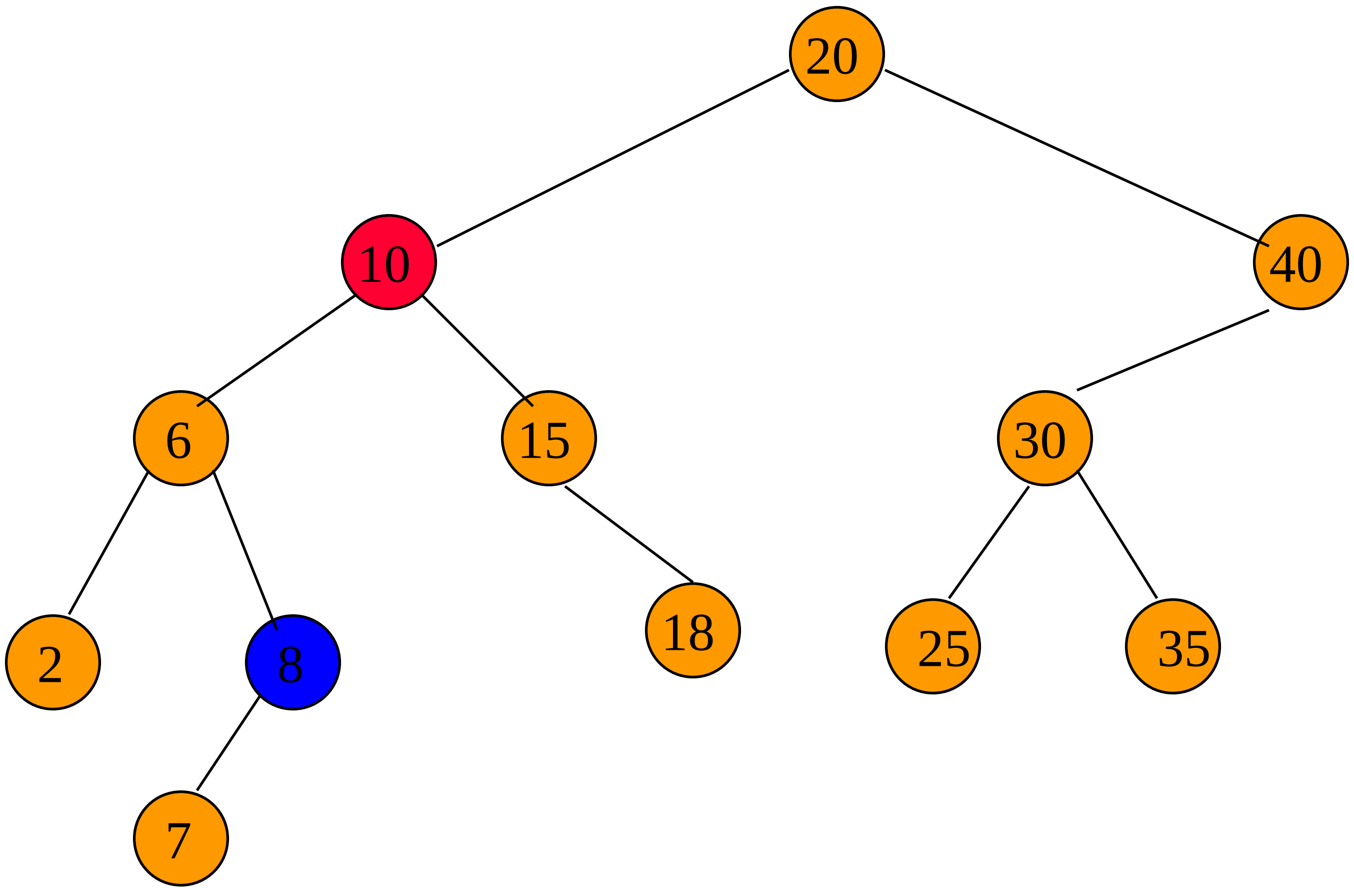


# Delete a Degree 2 Node (key:10)



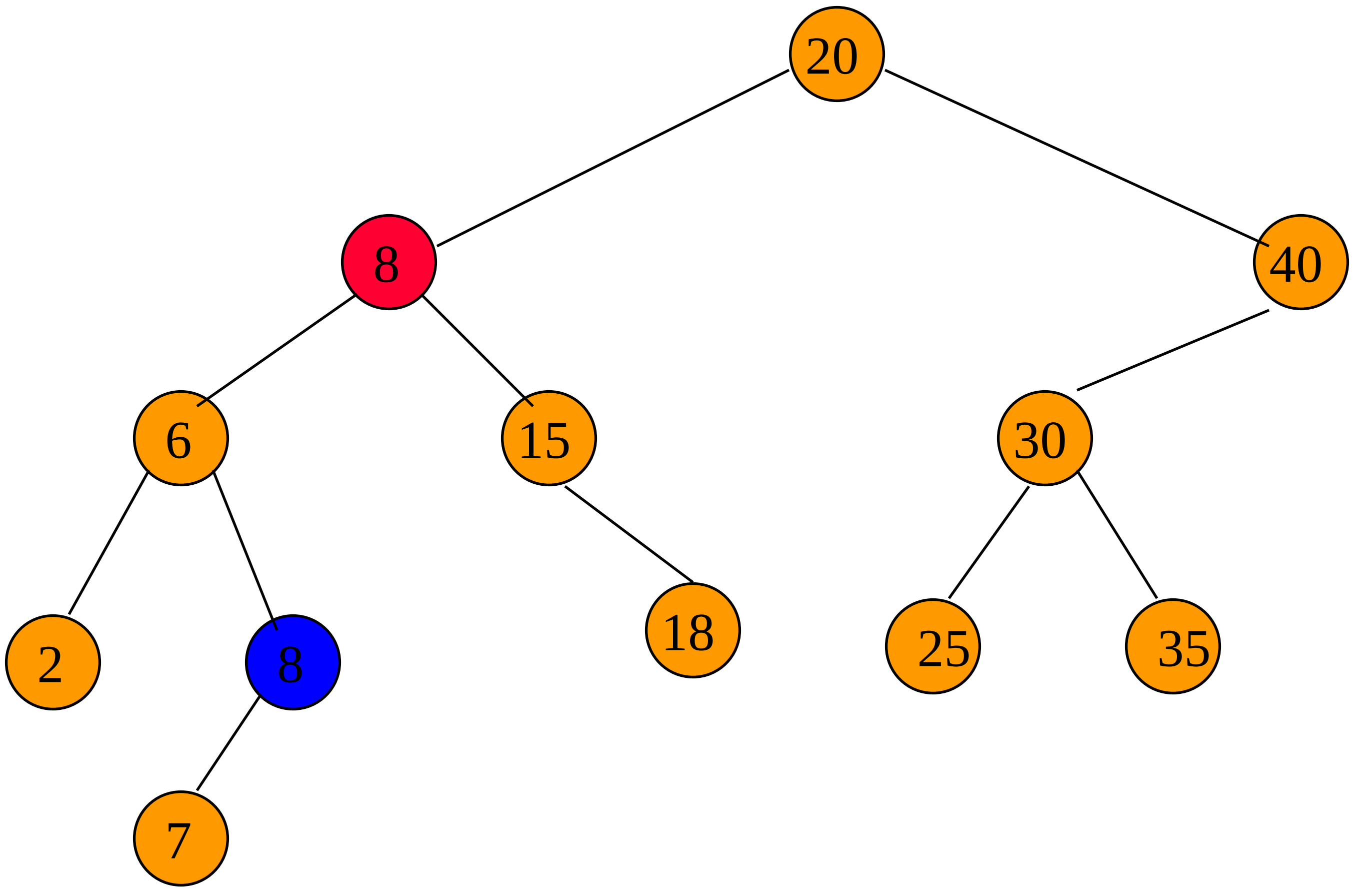

Replace with largest key in left subtree (or smallest in right subtree).

# Delete a Degree 2 Node (key:10)



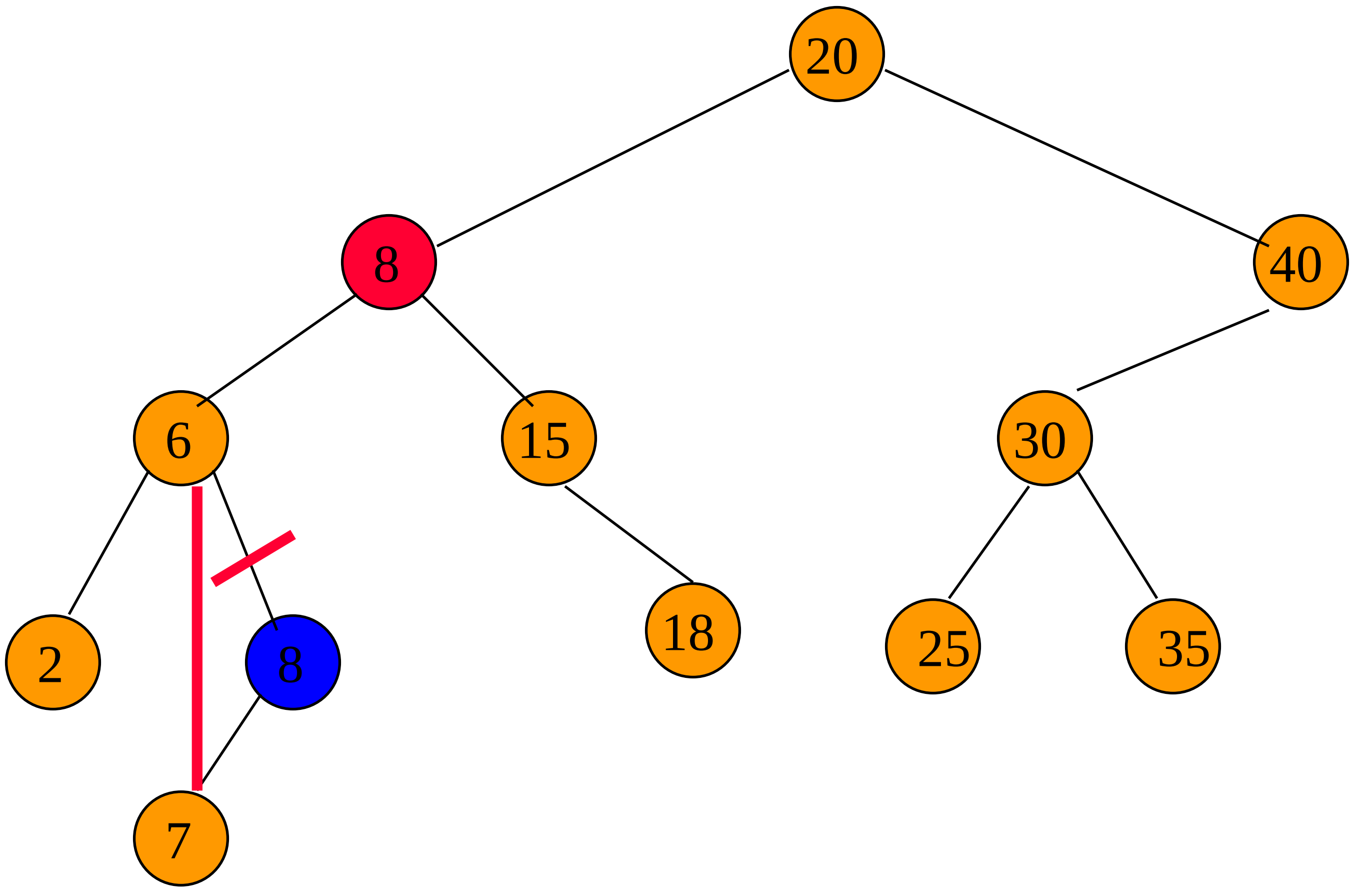

Replace with largest key in left subtree (or smallest in right subtree).

# Delete a Degree 2 Node (key:10)



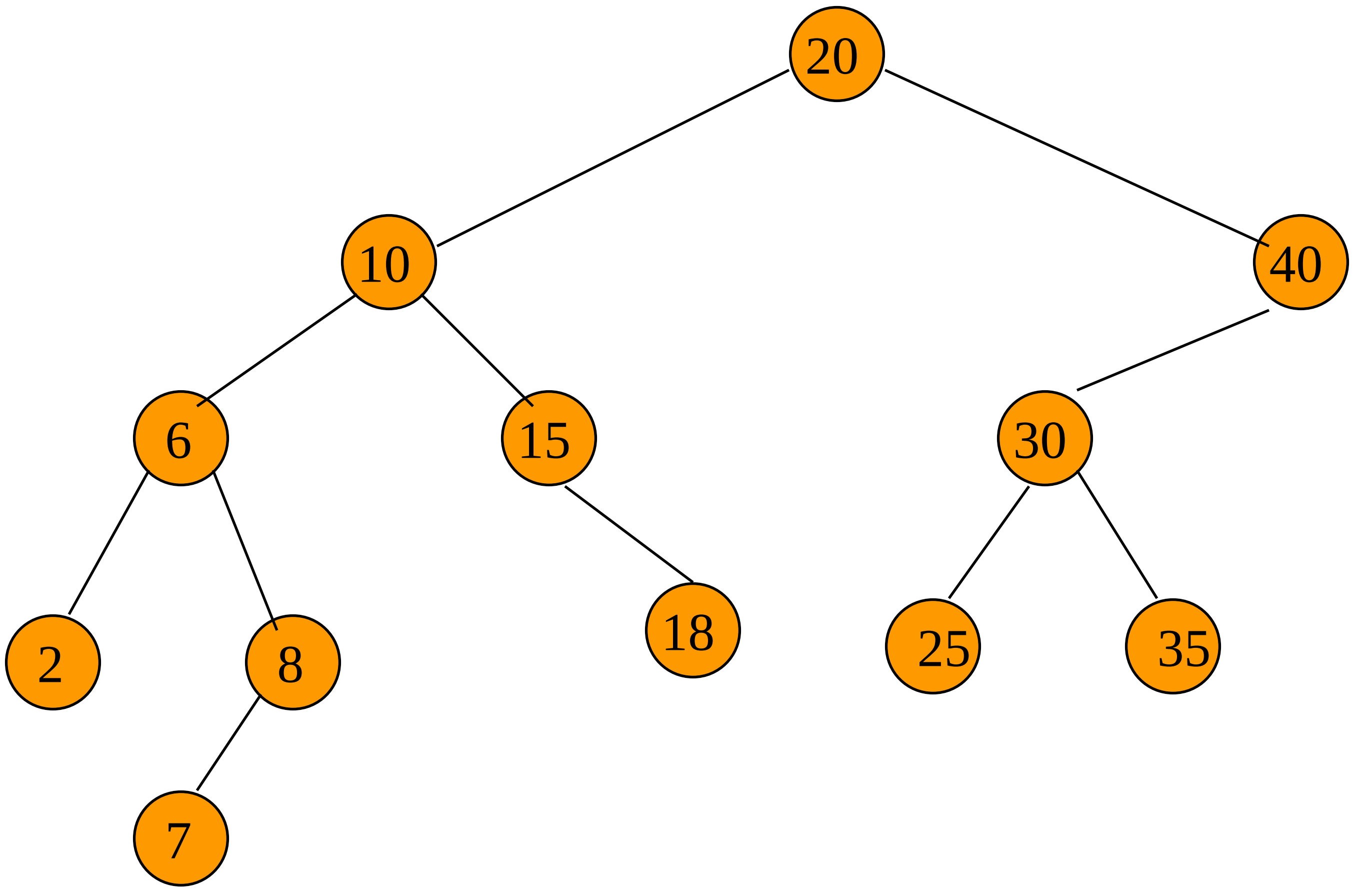

Replace with largest key in left subtree (or smallest in right subtree).

# Delete a Degree 2 Node (key:10)

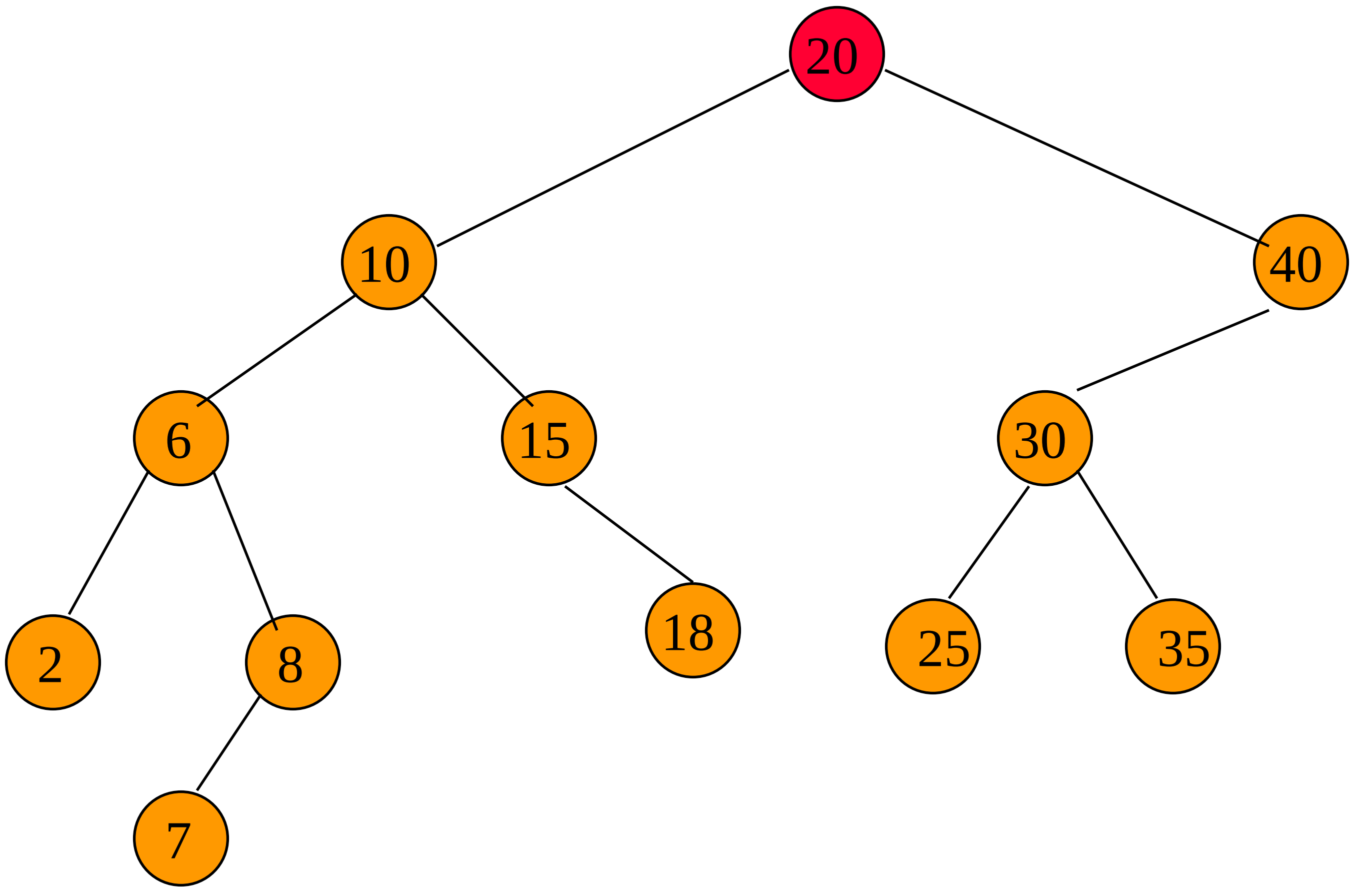



Largest key must be in a leaf or degree 1 node.

# Delete a Degree 2 Node (key:20)

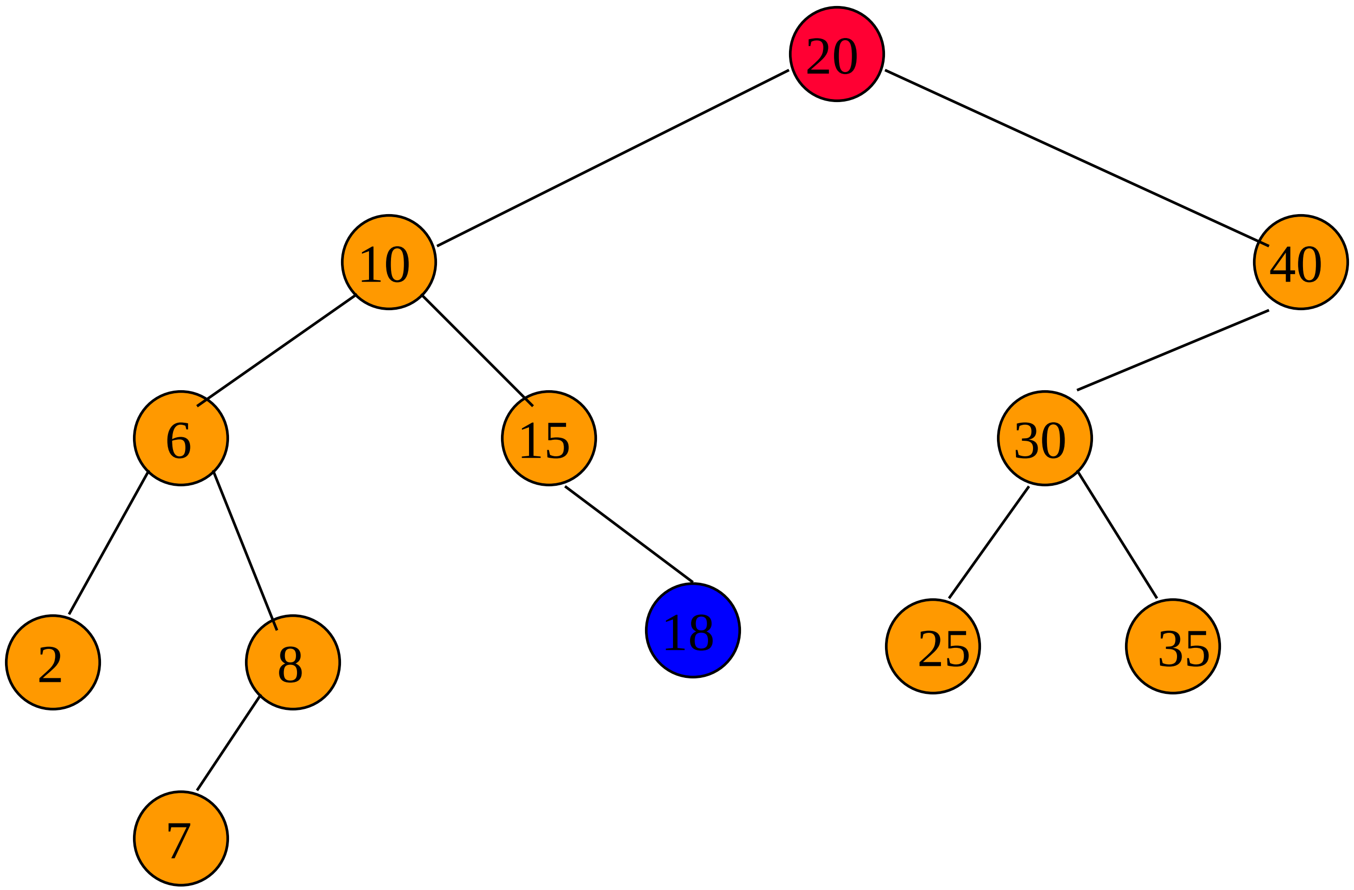


# Delete a Degree 2 Node



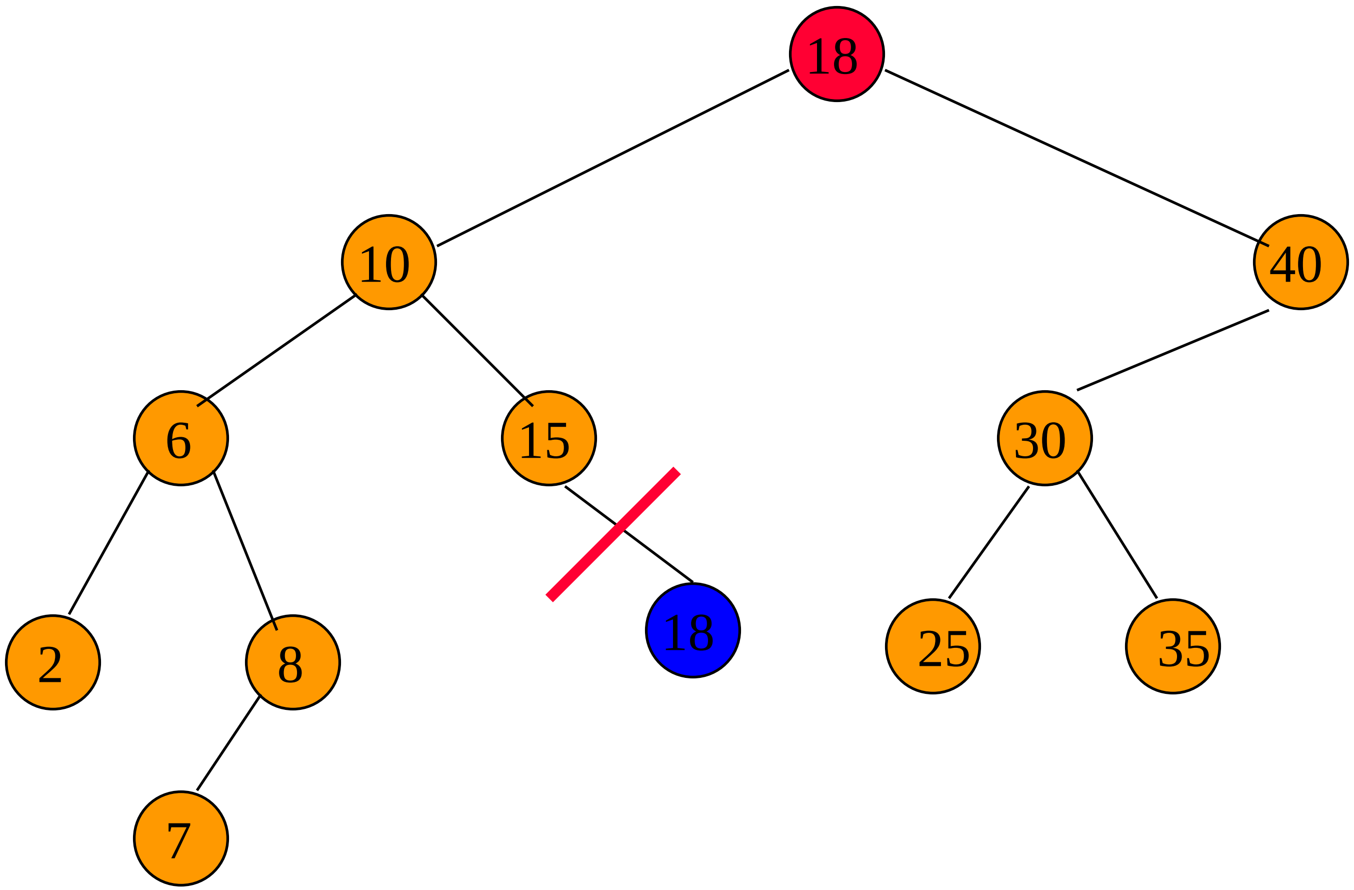

Replace with largest in left subtree.

# Delete a Degree 2 Node



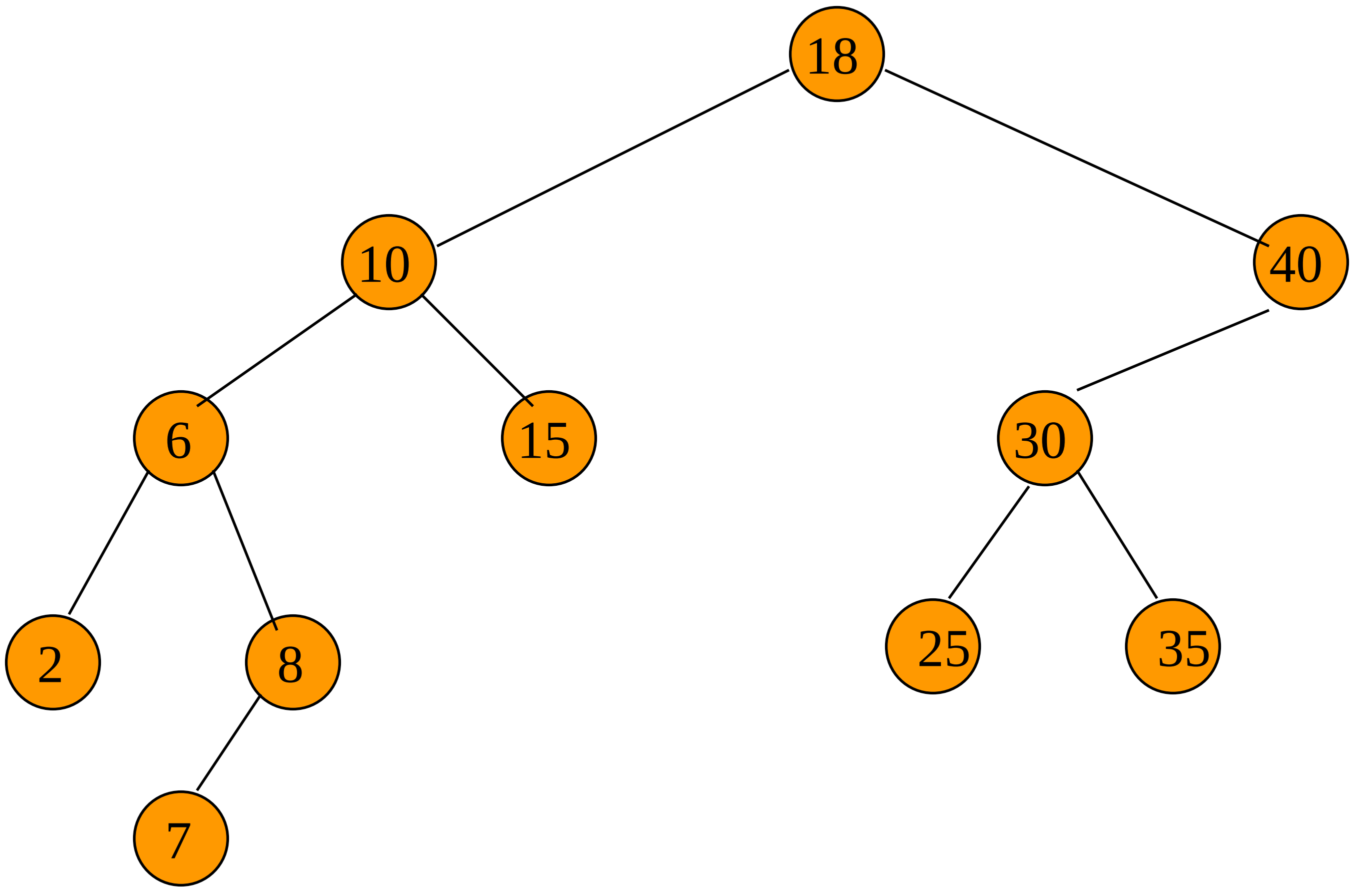

Replace with largest in left subtree.

# Delete a Degree 2 Node




Replace with largest in left subtree.

# Delete a Degree 2 Node




Complexity is O(height).

# BST Operations: Deletion

- Removes a specified item from the BST and adjusts the tree
- Uses a binary search to locate the target item:
  - starting at the root it probes down the tree till it finds the target or reaches a leaf node (target not in the tree)
- Removal of a node must not leave a ‘gap’ in the tree

# Deletion in BST - Pseudocode

**method delete(key)**

I if the tree is empty return false

II Attempt to locate the node containing the target using the binary search algorithm

if the target is not found return false

else the target is found, so remove its node:

Case 1: if the node has 2 empty subtrees (leaf node)

replace the link in the parent with null

Case 2: if the node has no left child

- link the parent of the node to the right (non-empty) subtree

# Deletion in BST - Pseudocode

Case 3: **if** the node has no right child

- link the parent of the target to the left (non-empty) subtree

Case 4: **if** the node has a left and a right subtree

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

# *Delete node having data as key*

```
void delete(ptrnode root, key) {
  ptrnode current= search(root, key);
  ptrnode toDelete = current;
  if (current != NULL) {          // Leaf or one child
    if (current->lchild == NULL) {
      current = current ->rchild;     // No left child
      free(toDelete);
    } else if (current ->rchild == NULL) { // No right child
      current = current ->lchild;
      free(toDelete);
    } else {                             // Two children
      successor = succ(curent);
      current->data = successor->data;
      delete(current ->rchild,successor->data);
    }
  }
}
```

# *Analysis of BST Operations*

- The complexity of operations **`search`**, **`insert`** and **`delete`** in BST is O(h), where h is the height.
- O(lgn) when the tree is balanced. The updating operations cause the tree to become unbalanced.
- The tree can degenerate to a linear shape and the operations will become O (n)

# *Better Search Trees*

Prevent the degeneration of the BST :

- A BST can be set up to maintain balance during updating operations (insertions and removals)
- Types of  Search Trees which maintain the optimal performance:
  - splay trees
  - AVL trees
  - 2-4 Trees
  - Red-Black trees
  - B-trees