AST20105 Data Structures & Algorithms

CHAPTER 6 – TREES I

Instructed by Garret Lai

Before Start

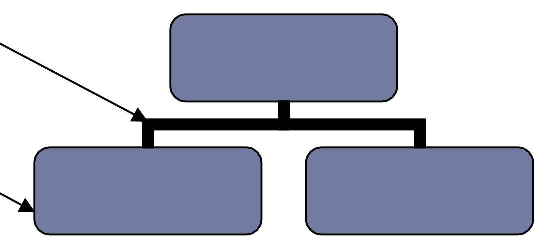
- Linked lists usually provide greater flexibility than arrays,
  - but they are linear structures and
  - it is difficult to use them to organize a hierarchical representation of objects.

Before Start

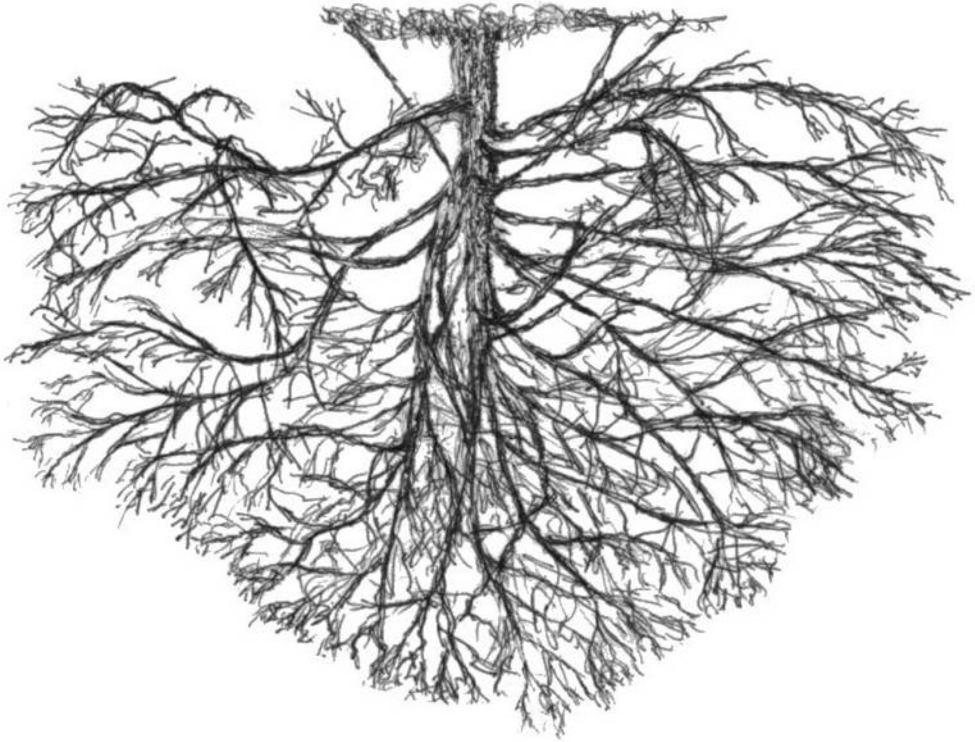
- Although stacks and queues reflect some hierarchy
- They are limited to only one dimension.

Before Start

- To overcome this limitation
  - We create a new data type called a tree that consists of nodes and arcs.

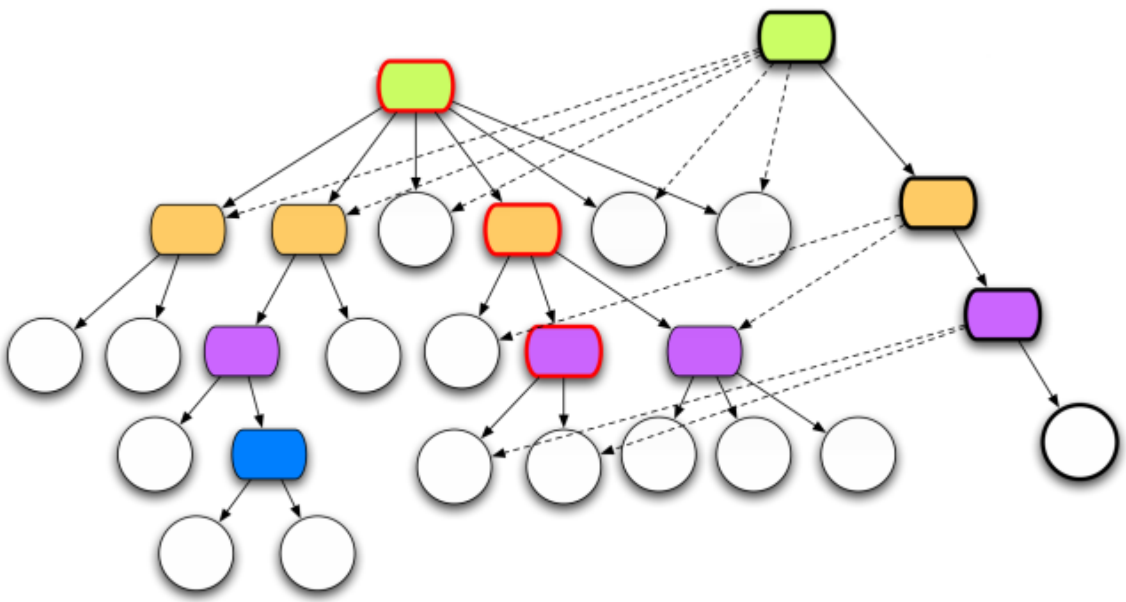


Trees



Definition

- A tree is a way to store data in hierarchical manner
- It contains a collection of nodes with no node cycle

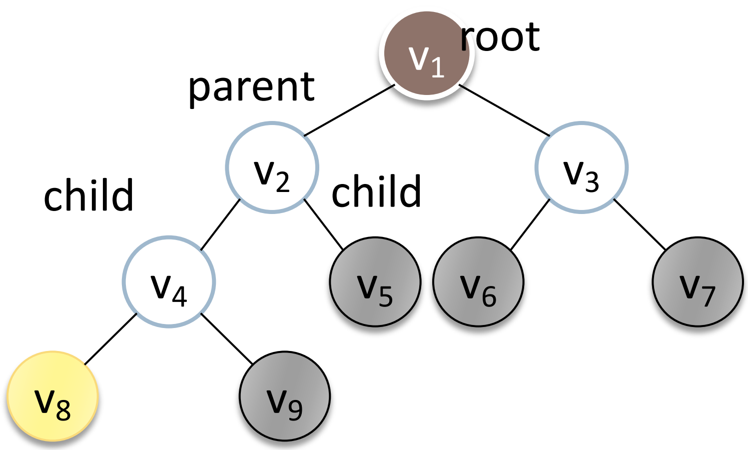


Definition

- Unlike natural trees, these trees are depicted upside down with
  - the root at the top and
  - the leaves (terminal nodes) at the bottom

Terminology

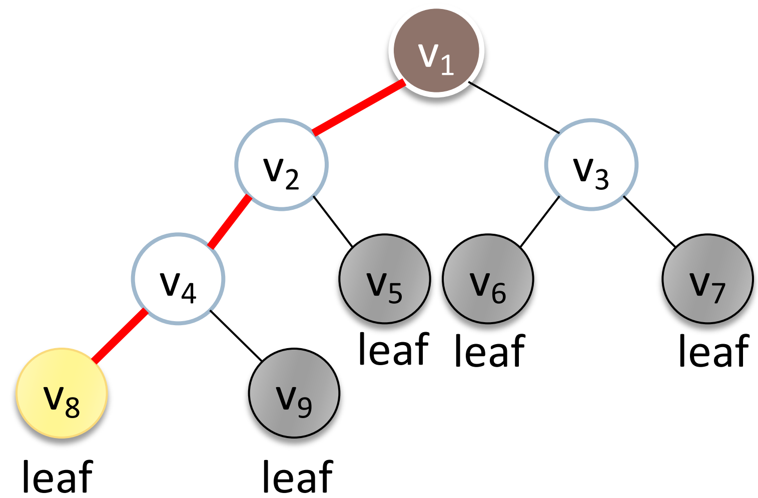
- Node / Vertex:
  - Basic element of a tree that used to store data and pointer(s) to other nodes / vertices
- Root:
  - The top node / starting node of the tree
- Parent and child:
  - Every node except the root has one parent
  - A node can have any number of children
- Sibling:
  - Nodes with same parent



- Root: v_1
- Parent and child: v_2 is the parent and v_4 and v_5 are the children
- Sibling: v_2 and v_3 are sibling, v_4 and v_5 are sibling, v_6 and v_7 are sibling, v_8 and v_9 are sibling

Terminology

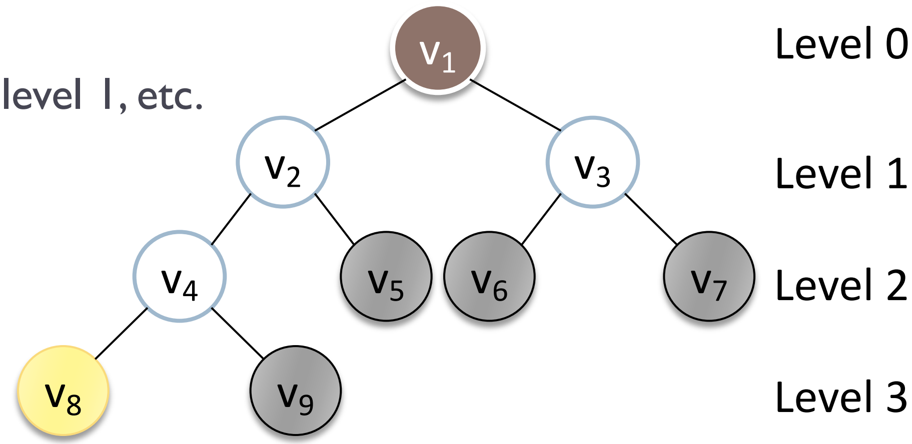
- Leaf:
  - Node that with no children
- Edge / Link:
  - A link from parent node to a child node
- Path:
  - A sequence of nodes, i.e. v_0, v_1, v_2, …, v_n, where there is an edge from one node to the next
- Length:
  - Number of edges on the path



- Leaf: v_5 , v_6 , v_7 , v_8 , v_9 are leaves
- Edge: All the lines are edge
- Path: (v_1, v_2, v_4, v_8) is a path from v_1 to v_8
- Length: The length of the path (v_1, v_2, v_4, v_8) is 3

Terminology

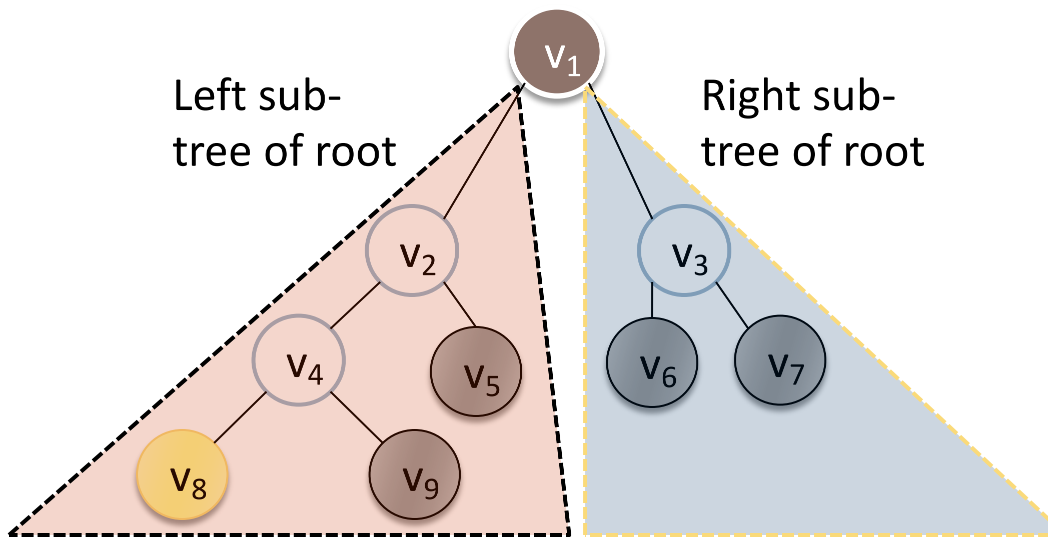
- Level:
  - Root is level 0, the children of root is level 1, etc.
- Node depth:
  - Length of the unique path from the root to the node
- Tree depth:
  - Depth of the deepest leaf
- Node height:
  - Length of the longest path from the node to a leaf
  - All leaves are at height 0
- Tree height:
  - Height of the root



- Level: v_1 is at level 0, v_2 & v_3 are at level 1, v_4, v_5, v_6 & v_7 are at level 2, v_8 & v_9 are at level 3
- Node depth of v_4: 2 (Since v_1 -> v_2 -> v_4)
- Tree depth: 3
- Node height of v_2: 2 (Since v_2 -> v_4 -> v_9)
- Tree height: 3 (Since v_1 -> v_2-> v_4 -> v_8)

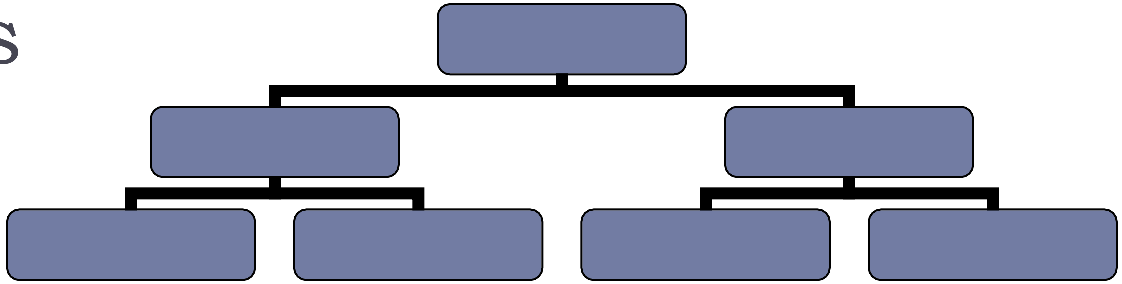
Terminology

- Ancestor:
  - The parent, grand parent, grand grand parent, … of a node
- Descendant:
  - The children, grand children, grand grand children, … of a node
- Left sub-tree and right sub-tree
  - Smaller tree consisting of a node and all its descendants



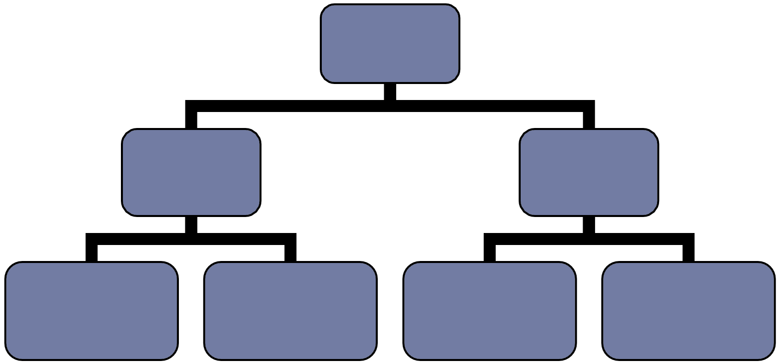
- Ancestor of v_9: v_4, v_2, v_1
- Descendant of v_2: v_4, v_5, v_8, v_9
- Left sub-tree of v_1: Purple triangle
- Right sub-tree of v_1: Blue triangle

Binary Trees



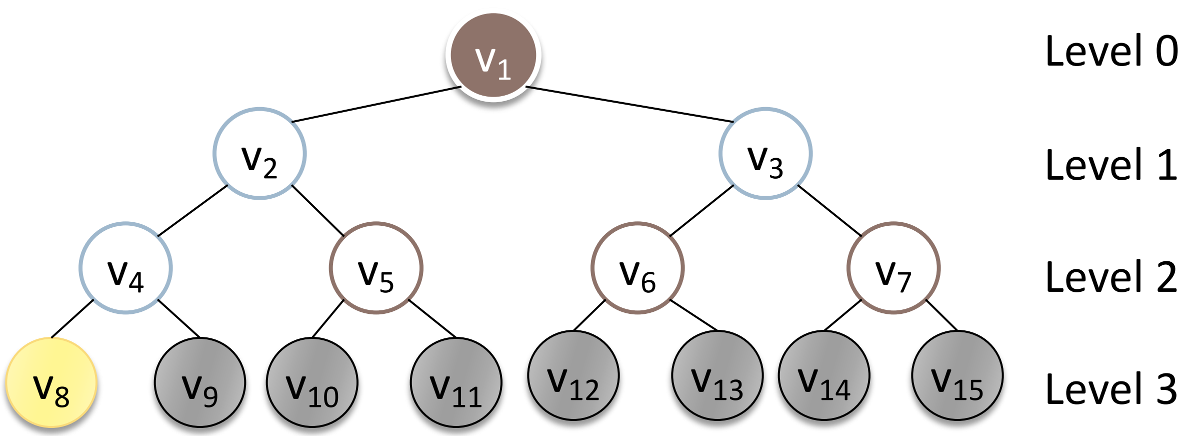
Binary Tree

- The simplest form of tree is a binary tree.
- A binary tree is a tree in which every node in the tree can have AT MOST two children
- A binary tree consists of
  - a *node* (called the **root** node) and
  - left and right sub-trees.
    - Both the sub-trees are themselves binary trees.



Full Binary Tree

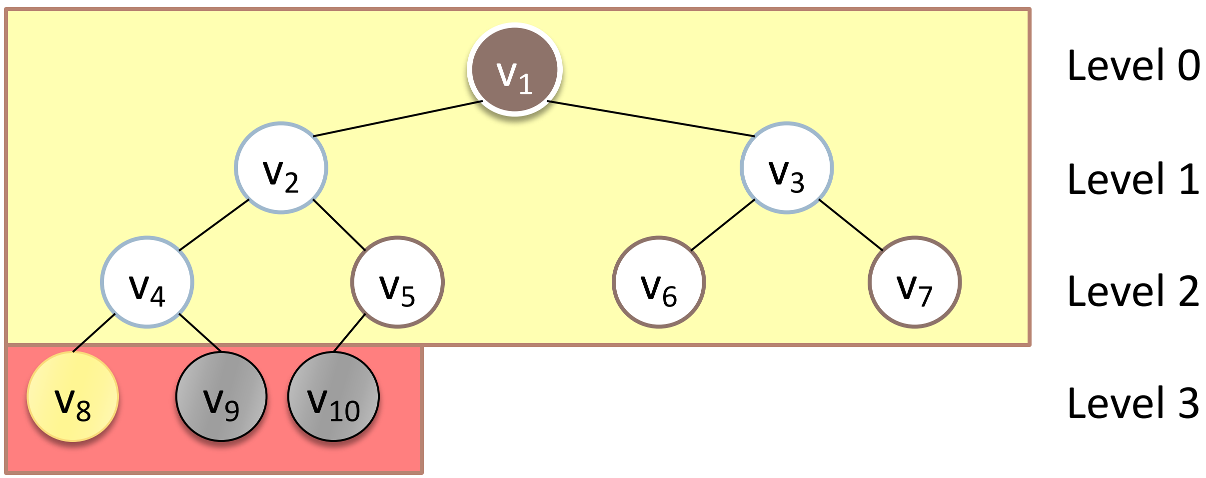
- A binary tree with 2^i nodes at level i
- Properties:
  - Total number of node in the tree = $2^{h+1} - 1$, where h is the tree height



- Total no. of nodes at level 0 = $2^0 = 1$
- Total no. of nodes at level 1 = $2^1 = 2$
- Total no. of nodes at level 2 = $2^2 = 4$
- Total no. of nodes at level 3 = $2^3 = 8$
- Total no. of nodes in the tree (h = 3) = $2^{(3+1)} - 1 = 15$

Complete Binary Tree

- A binary tree of height h having complete filled to depth h-1 and at depth h, filled nodes are on the left

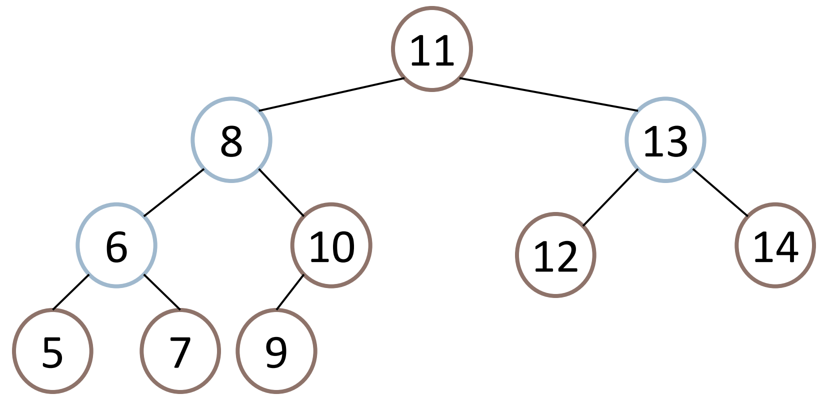


- A binary tree of height 3
- Complete filled to depth 3 – 1 = 2
- At depth 3, filled nodes are on the left

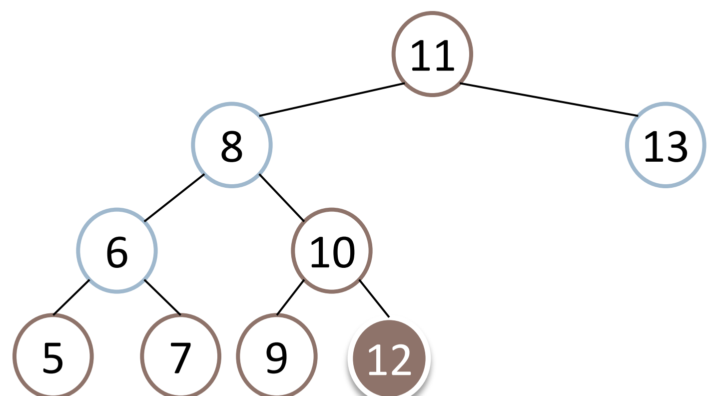
Binary Search Tree (BST)

- Binary Search Tree is a binary tree that stores values / keys in a way such that insertion, deletion and searching could be done efficiently

- Properties:
  - For every node v, all the values in its left sub-tree are SMALLER than the value in v, and all the values in its right sub-tree are LARGER than the value in v

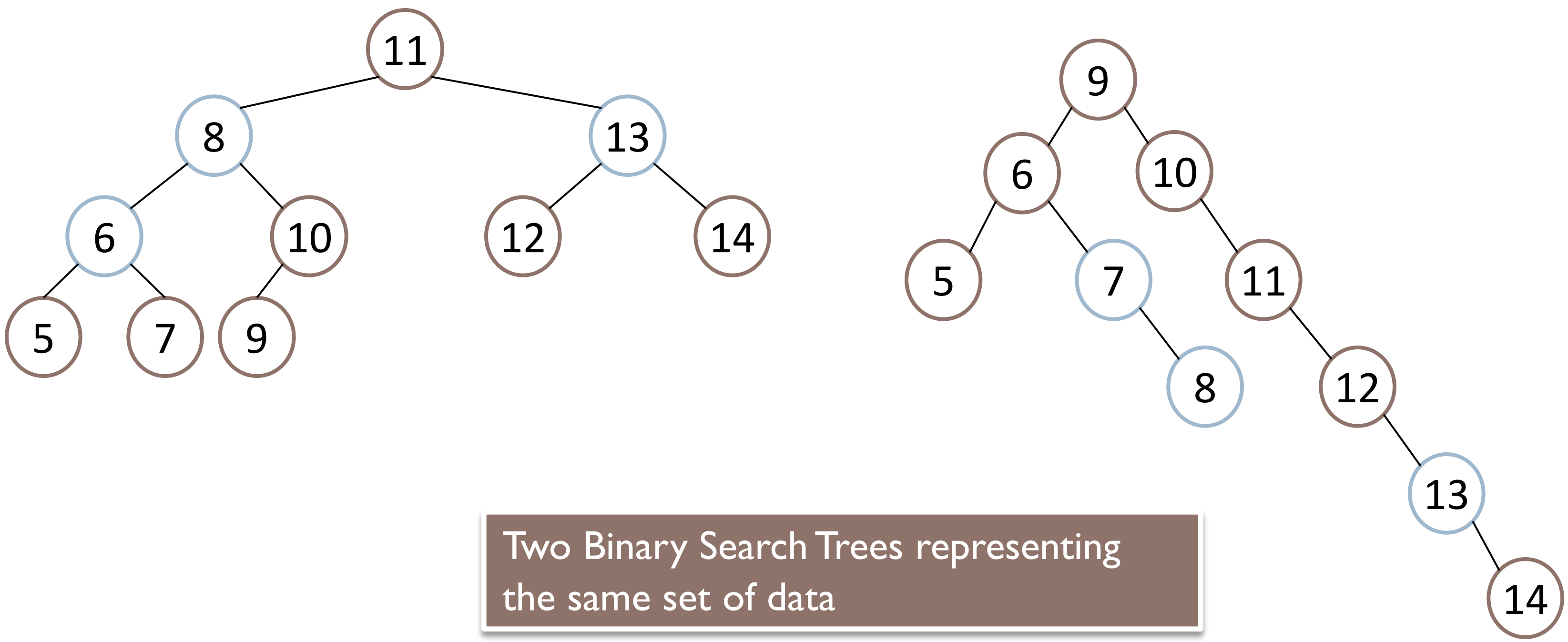


A Binary Search Tree



NOT a Binary Search Tree

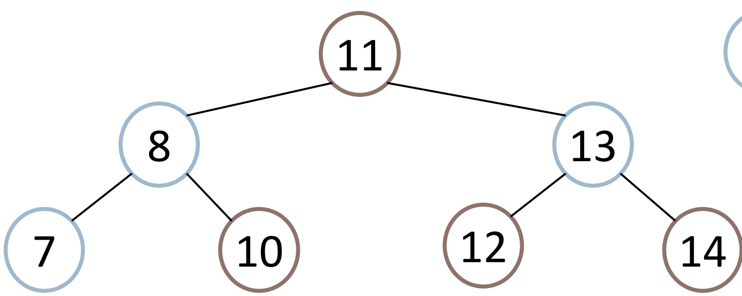
Binary Search Tree (BST) - Example

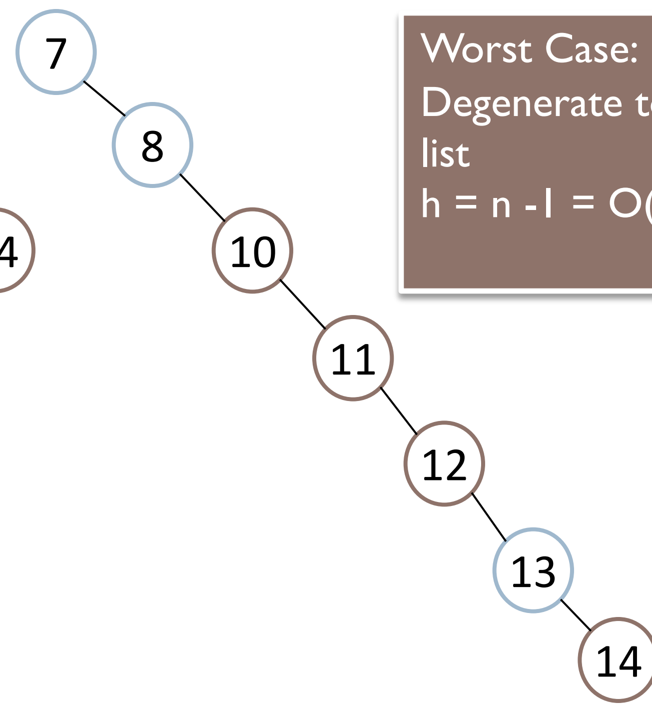


Two Binary Search Trees representing the same set of data

Height of Binary Search Tree

- From the last slide, it is easy to observe that a same set of data could be stored as different Binary Search Tree
- The height of a Binary Search Tree could vary depending on the order of insertion
- Say, we got n data to store in a Binary Search Tree





Worst Case:
Degenerate to a linked list
$h = n - 1 = O(n)$

Best Case: Balanced Tree
$2^{h+1} - 1 = n$
$2^{h+1} = n + 1$
$\log_2(2^{h+1}) = \log_2(n+1)$
$h+1 = \log_2(n+1)$
$h = \log_2(n+1) - 1 = O(\log n)$

Implementing Binary Search Trees

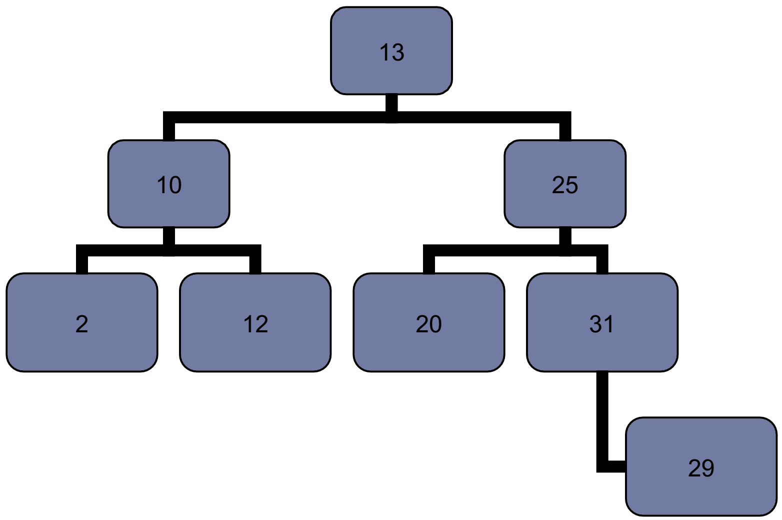
Implementing Binary Search Trees

- Binary trees can be implemented in at least two ways:
  - As arrays and
  - As linked structures.

Implementing Binary Search Trees

- To implement a tree as an array,
  - A node is declared as a structure with an information field and two "pointer" fields
  - These pointer fields contain the indexes of the array cells in which the left and right children are stored, if there are any.
  - The root is always located in the first cell, cell 0, and -1 indicates a null child.

Implementing Binary Search Trees



Index	Info	Left	Right
0	13	4	2
1	31	6	-1
2	25	7	1
3	12	-1	-1
4	10	5	3
5	2	-1	-1
6	29	-1	-1
7	20	-1	-1

Implementing Binary Search Trees

- However, this implementation may be inconvenient, even if the array is flexible
  - Locations of children must be known to insert a new node, and these locations may need to be located sequentially.

Implementing Binary Search Trees

- However, this implementation may be inconvenient, even if the array is flexible
  - After deleting a node from the tree, a hole in the array would have to be eliminated.
    - This can be done either by using a special marker for an unused cell, which may lead to populating the array with many unused cells, or
    - By moving elements by one position, which also requires updating references to the elements that have been moved.

Implementing Binary Search Trees

- A Binary Search Tree is constructed using nodes, similar to linked list
- A Binary Search Tree node / vertex should contain
  - Data
  - Two pointers (left pointer to the left sub-tree and right pointer to the right sub-tree)
- A leaf node has both left and right pointers point to NULL

```
class BSTNode
{
  public:
    BSTNode(const int value)
    {
       left = right = NULL;
       data = value;
    }

    int data;
    BSTNode* left;
    BSTNode* right;
};
```

Binary Search Tree Operations

- BSTNode* insertNode(BSTNode*& n, const int v)
  - Insert a node to the Binary Search Tree
- BSTNode* deleteNode(BSTNode*& n, const int v)
  - Delete a node from the Binary Search Tree
- BSTNode* findNode(BSTNode* n, const int v)
  - Find the node with value v
- BSTNode* findMin(BSTNode* n)
  - Find the node with the smallest key
- BSTNode* findMax(BSTNode* n)
  - Find the node with the largest key
- void preOrder(BSTNode* n)
  - Print all the keys using pre-order traversal
- void inOrder(BSTNode* n)
  - Print all the keys using in-order traversal
- void postOrder(BSTNode* n)
  - Print all the keys using post-order traversal

Implementing Binary Search Trees

BSTree.h

```
class BSTree
{
   public:
     BSTNode* root;
     BSTNode* temp;

     BSTree();
     ~BSTree();
     BSTNode* insertNode(BSTNode*& n, const int v);
     BSTNode* deleteNode(BSTNode*& n, const int v);
     BSTNode* findNode(BSTNode* n, const int v);
     BSTNode* findMin(BSTNode* n);
     BSTNode* findMax(BSTNode* n);
     void preOrder(BSTNode* n);
     void inOrder(BSTNode* n);
     void postOrder(BSTNode* n);
};
```

```
BSTree::BSTree()
{
   root = NULL;
   temp = NULL;
}

BSTree::~BSTree()
{
}
```

BSTree.cpp

Searching a Binary Search Tree

Searching in BST

- Search algorithm traverses the tree "in-depth", choosing appropriate way to go, following binary search tree property and compares value of each visited node with the one, we are looking for.

- Algorithm stops in two cases:
  - a node with necessary value is found;
  - algorithm has no way to go.

Searching in BST

- Search algorithm in detail
  - Search algorithm utilizes recursion.
  - Starting from the root,

Searching in BST

- Search algorithm in detail

1. check, whether value in current node and searched value are equal. If so, **value is found.** Otherwise,

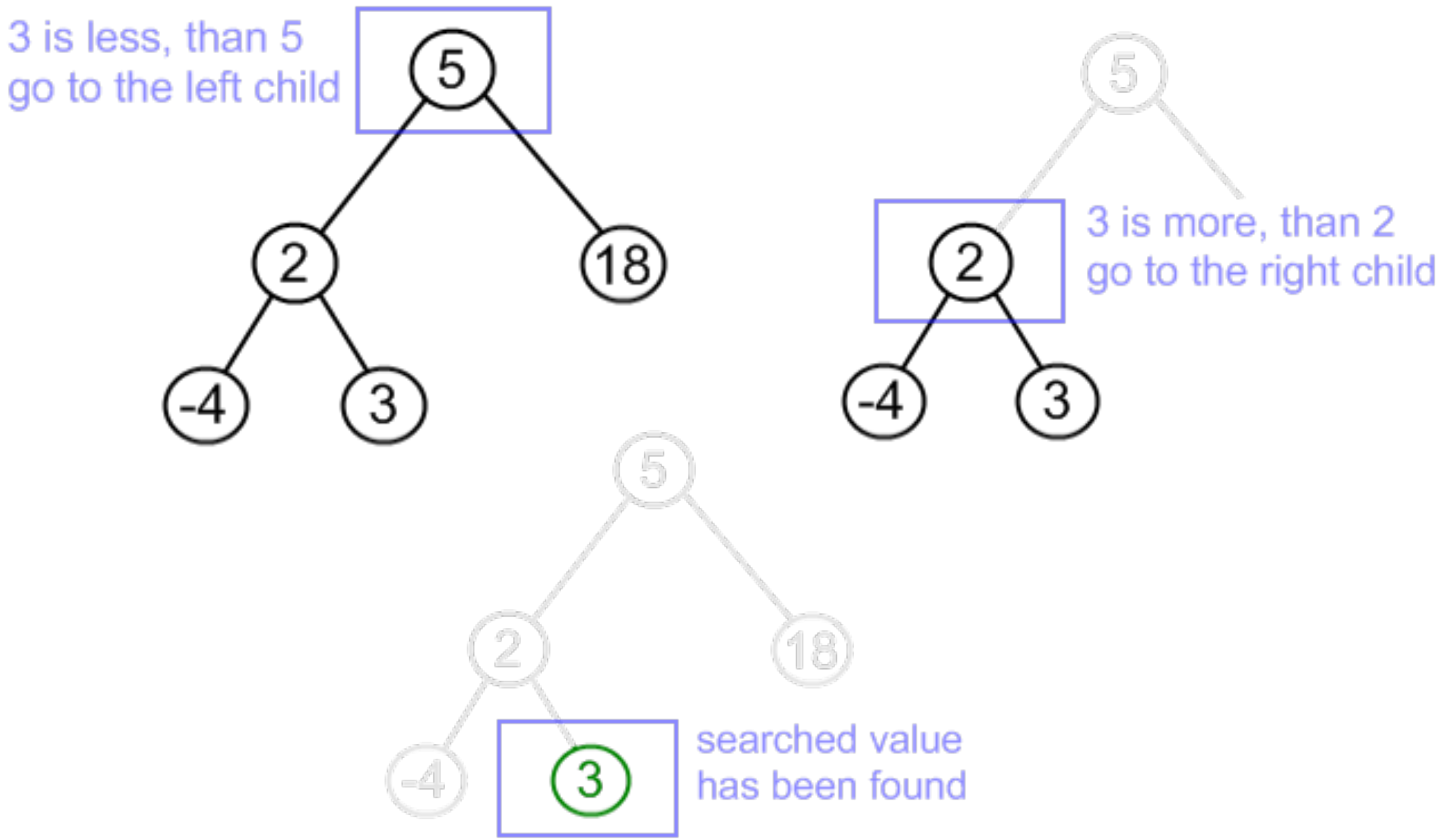
2. if searched value is less than the node's value:
   - if current node has no left child,
     searched value doesn't exist in the BST;
   - otherwise, handle the left child with the same algorithm.

3. if searched value is greater than the node's value:
   - if current node has no right child,
     searched value doesn't exist in the BST;
   - otherwise, handle the right child with the same algorithm.

- Running time: O(h)

Searching in BST

- Search for 3 in the tree



Find Min / Max

- findMin
  - Algorithm:
    - Start at the root and keep going left if there is a left child. The ending point is the smallest element
  - Running time: O(h)

- findMax
  - Algorithm:
    - Start at the root and keep going right if there is a right child. The ending point is the largest element
  - Running time: O(h)

```cpp
BSTNode* BSTree::findNode(BSTNode* n, const int v) {
    if(n == NULL)
        return NULL;
    if(v < n->data)
        return findNode(n->left, v);
    else if(v > n->data)
        return findNode(n->right, v);
    else
        return n;
}

BSTNode* BSTree::findMin(BSTNode* n) {
    if(n == NULL)
        return NULL;
    else if(n->left == NULL)
        return n;
    else
        return findMin(n->left);
}

BSTNode* BSTree::findMax(BSTNode* n) {
    if(n == NULL)
        return NULL;
    else if(n->right == NULL)
        return n;
    else
        return findMax(n->right);
}
```

BSTree.cpp

Tree Traversal

Tree Traversal

- Tree traversal is the process of visiting each node in the tree exactly one time.

- Traversal may be interpreted as putting all nodes on one line or linearizing a tree.

Tree Traversal

- The definition of traversal specifies only one condition
  - Visiting each node only one time
  - But it does not specify the order in which the nodes are visited

Tree Traversal

- Hence, there are as many tree traversals as there are permutations of nodes

- For a tree with n nodes, there are n! different traversals.

- Most of them, however, are rather chaotic and do not indicate much regularity so that implementing such traversals lacks generality.

Tree Traversal

- In the face of such an abundance of traversals and the apparent uselessness of most of them
  - We would like to restrict our attention to two classes only, namely
    - Breadth-first traversals and
    - Depth-first traversals

Breadth-first Traversal

- Breadth-first traversal is visiting each node
  - starting from the highest (or lowest) level and
  - moving down (or up) level by level,
  - visiting nodes on each level from left to right (or from right to left).

Breadth-first Traversal

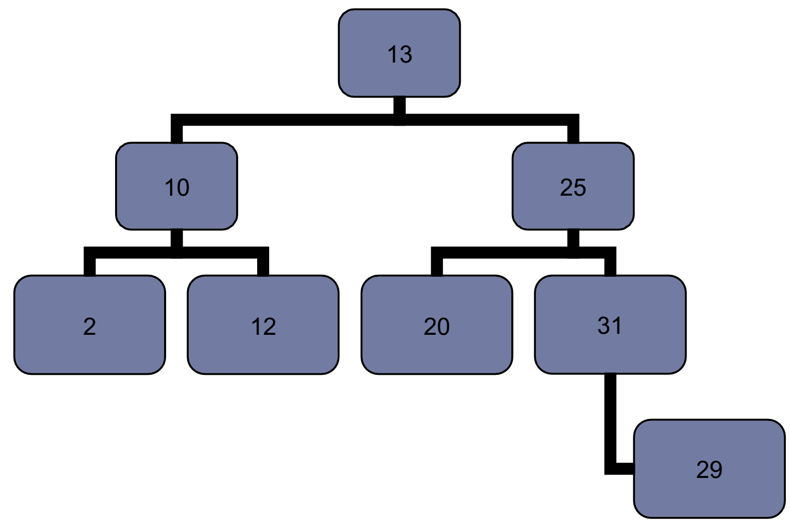
```
void BSTree::BreadthFirstTraversal(BSTNode *root) {
  if (root == NULL)
       return;
  deque <BSTNode *> queue;
  queue.push_back(root);  //push element at the back of the queue

  while (!queue.empty()) {
       BSTNode *p = queue.front();
       cout << p->data << "\t";
      queue.pop_front();

      if (p->left != NULL)
         queue.push_back(p->left);
      if (p->right != NULL)
         queue.push_back(p->right);
   }
   cout << endl;
}
```

Breadth-first Traversal

- **A top-down, left-to-right breadth-first traversal of the tree results in the sequence**
  - 13, 10, 25, 2, 12, 20, 31, 29



Depth-first Traversal

- Depth-first traversal
  - proceeds as far as possible to the left (or right),
  - then backs up until the first crossroad,
    goes one step to the right (or left), and
  - again as far as possible to the left (or right).
  - We repeat this process until all nodes are visited.

Depth-first Traversal

- This definition, however, does not clearly specify exactly when nodes are visited:
  - Before proceeding down the tree or after backing up?
  - There are some variations of the depth-first traversal.

Depth-first Traversal

- There are three tasks of interest in this type of traversal:
  - V – visiting a node
  - L – traversing the left subtree
  - R – traversing the right subtree

Depth-first Traversal

- An orderly traversal takes place if these tasks are performed in the same order for each node.

- The three tasks can themselves be ordered in 3! = 6 ways, so there are six possible ordered depth-first traversals:

  - VLR VRL
  - LVR RVL
  - LRV RLV

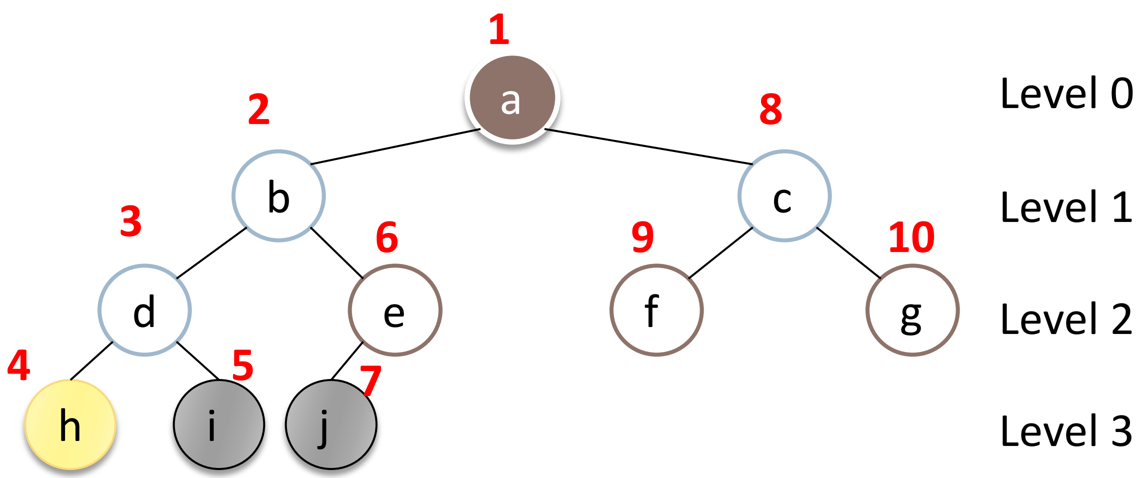
Depth-first Traversal

- It can be reduced to three traversals where the move is always from left to right and attention is focused on the following three traversals:

  - VLR – preorder tree traversal
  - LVR – inorder tree traversal
  - LRV – postorder tree traversal

Pre-order Traversal

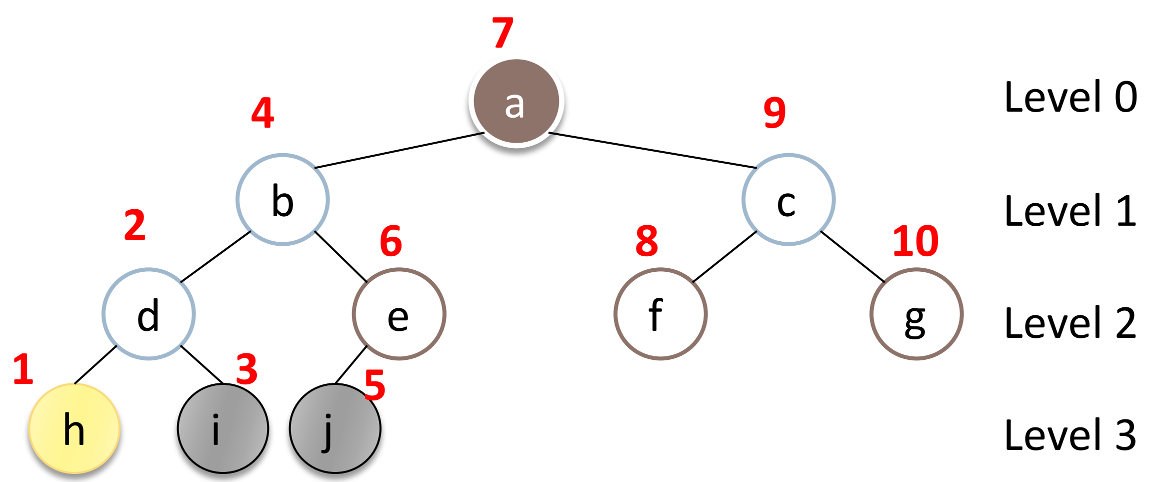
- Visit the node
- Recursively visit all nodes in the left sub-tree
- Recursively visit all nodes in the right sub-tree



Order to visit: a, b, d, h, I, e, j, c, f, g

In-order Traversal

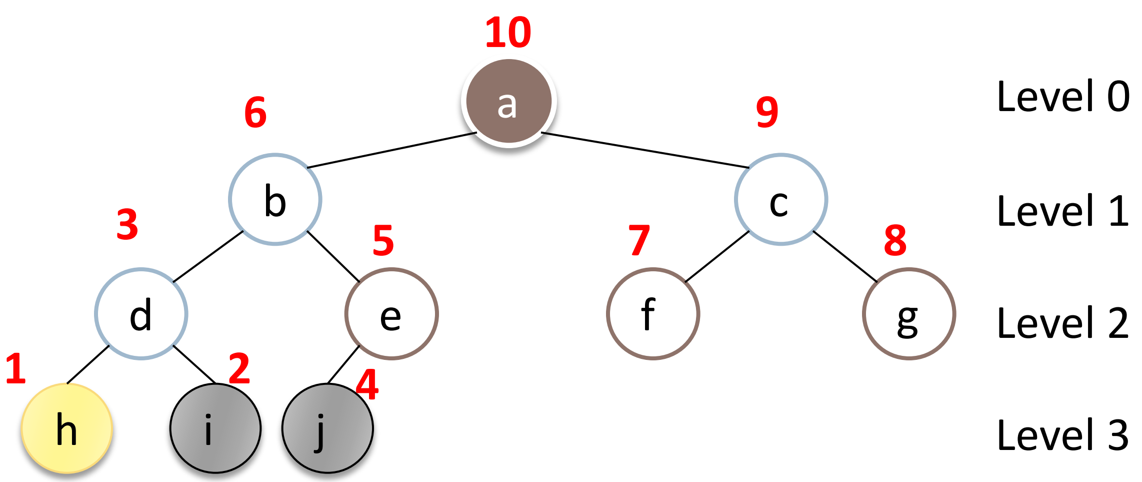
- Recursively visit all nodes in the left sub-tree
- Visit the node
- Recursively visit all nodes in the right sub-tree



Order to visit: h, d, i, b, j, e, a, f, c, g

Post-order Traversal

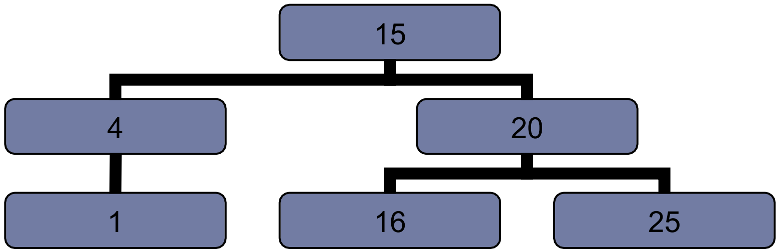
- Recursively visit all nodes in the left sub-tree
- Recursively visit all nodes in the right sub-tree
- Visit the node



Order to visit: h, I, d, j, e, b, f, g, c, a

Depth-first Traversal

- Exercise:
  - Pre-order:
  - In-order:
  - Post-order:



```
void BSTree::preOrder(BSTNode* n) {
    if(n != NULL) {
        cout << n->data << "\t";
        preOrder(n->left);
        preOrder(n->right);
    }
}

void BSTree::inOrder(BSTNode* n) {
    if(n != NULL) {
        inOrder(n->left);
        cout << n->data << "\t";
        inOrder(n->right);
    }
}

void BSTree::postOrder(BSTNode* n) {
    if(n != NULL) {
        postOrder(n->left);
        postOrder(n->right);
        cout << n->data << "\t";
    }
}
```

BSTree.cpp

Insertion

Insertion

- Adding a value to BST can be divided into two stages:
  - search for a place to put a new element;
  - insert the new element to this place.

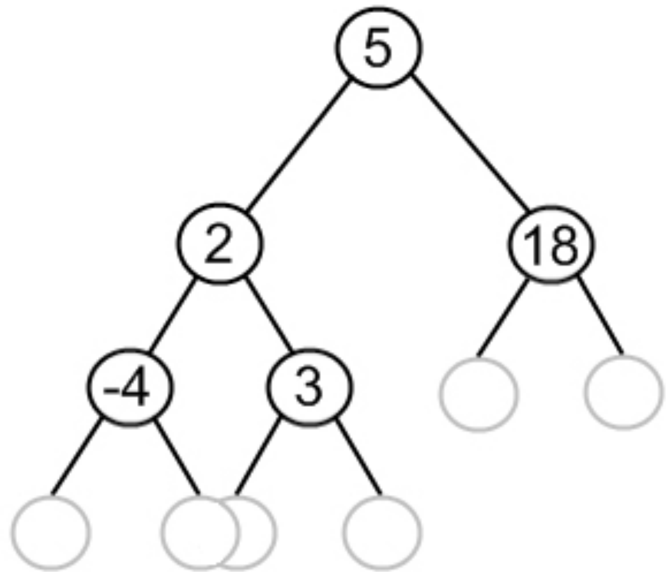
Insertion

- Search for a place
  - At this stage an algorithm should follow binary search tree property.
    - If a new value is less than the current node's value, go to the left subtree, else go to the right subtree.
    - Following this simple rule, the algorithm reaches a node, which has no left or right subtree.
    - By the moment a place for insertion is found, we can say for sure, that a new value has no duplicate in the tree.
    - Initially, a new node has no children, so it is a leaf.

Insertion

- Search for a place

  - Let us see it at the picture.
    Gray circles indicate possible places for a new node



Insertion

- Let's go down to algorithm itself. Starting from the root,

1. check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,

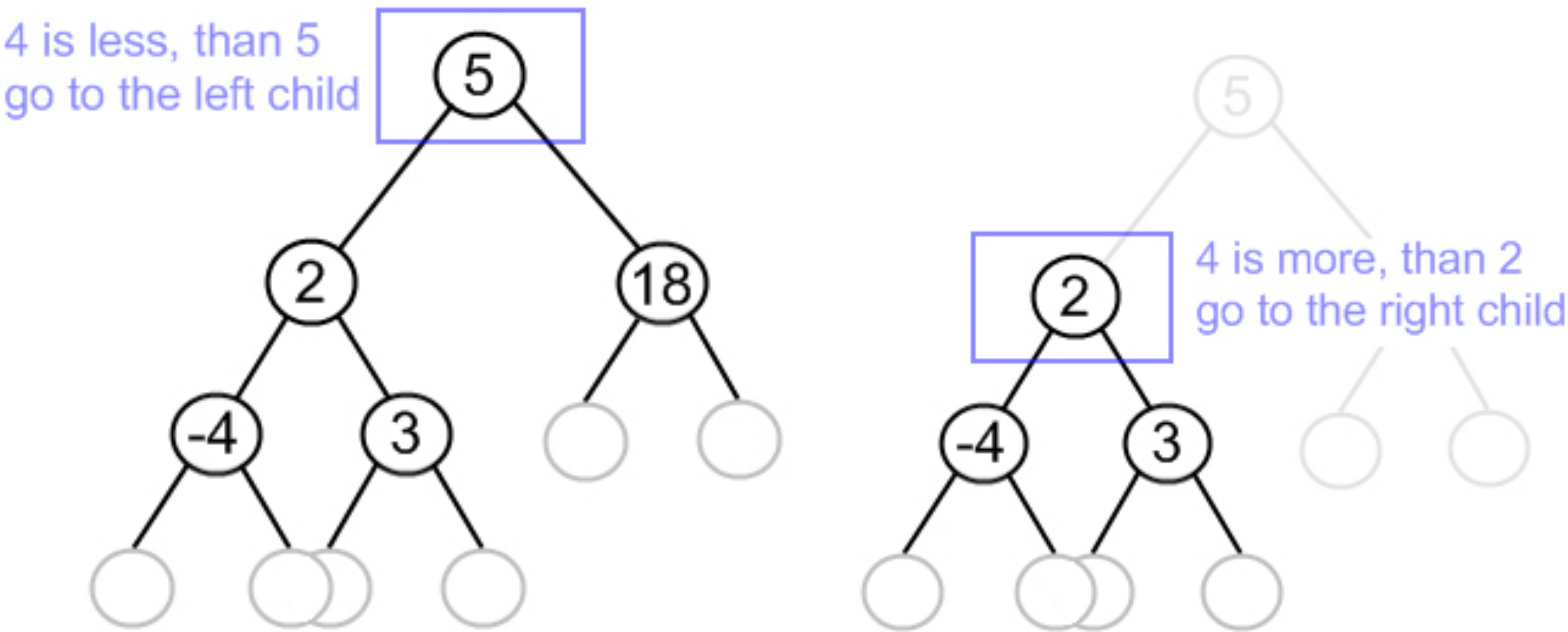
2. if a new value is less than the node's value:
    - if a current node has no left child, place for insertion has been found;
    - otherwise, handle the left child with the same algorithm.

3. if a new value is greater than the node's value:
    - if a current node has no right child, place for insertion has been found;
    - otherwise, handle the right child with the same algorithm.

- Running time: O(h)

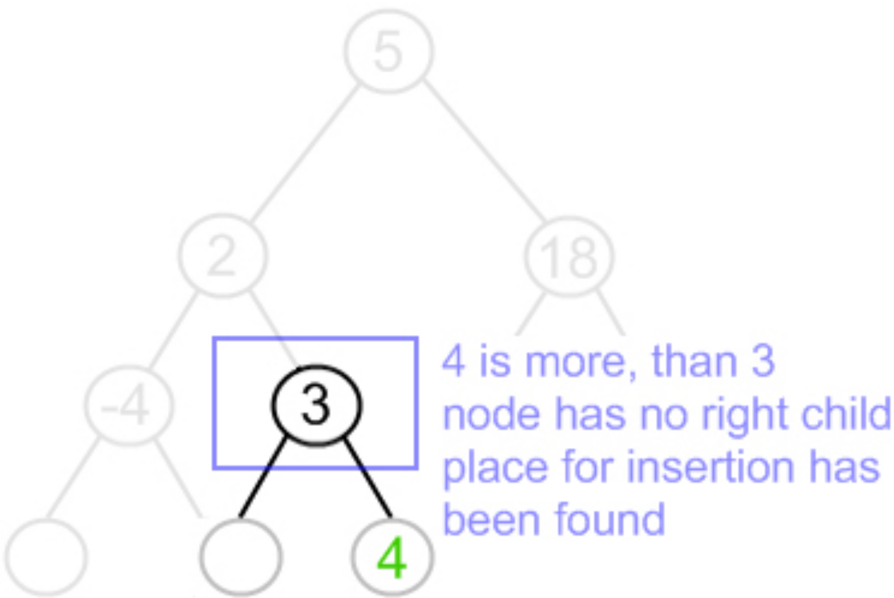
Insertion

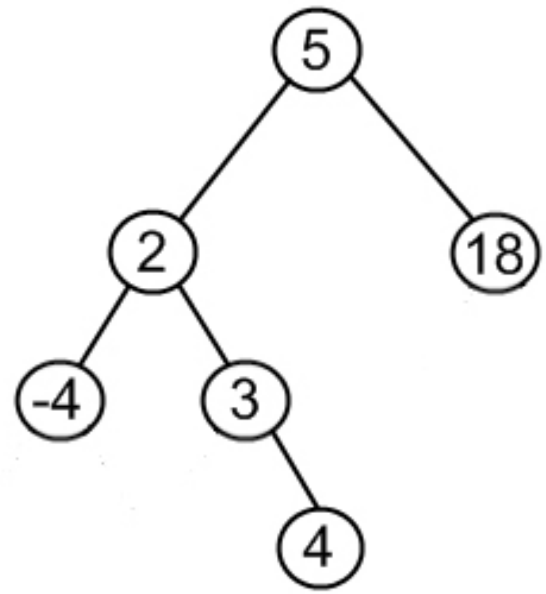
- Insert 4 to the tree, shown above



Insertion

- Insert 4 to the tree, shown above





Insertion

```
BSTNode* BSTree::insertNode(BSTNode*& n,const int v)
{
    if(n == NULL)
    {
        BSTNode* item = new BSTNode(v);
        n = item;
    }
    else
    {
        if(v < n->data)
            n->left = insertNode(n->left, v);
        else if(v > n->data)
            n->right = insertNode(n->right, v);
    }
    return n;
}
```

BSTree.cpp

Deletion

Deletion

- Remove operation on binary search tree is more complicated, than add and search.

- Basically, it can be divided into two stages:

  - search for a node to remove;

  - if the node is found, run remove algorithm.

Deletion

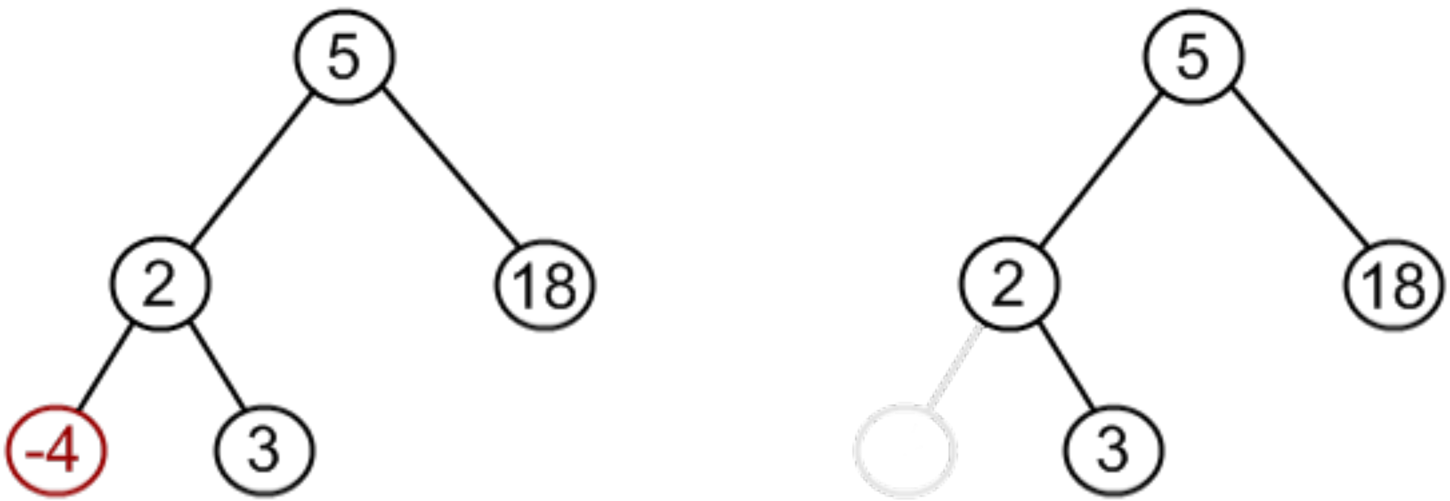
- Remove algorithm in detail
  - First stage is identical to algorithm for lookup, except we should track the parent of the current node.
  - Second part is more tricky.
    There are three cases, which are described below.

Deletion

- Case 1: Node to be removed has no children.
  - This case is quite simple.
  - Algorithm sets corresponding link of the parent to NULL and disposes the node.

Deletion

- Case 1: Node to be removed has no children.
  - **Example.** Remove -4 from a BST.



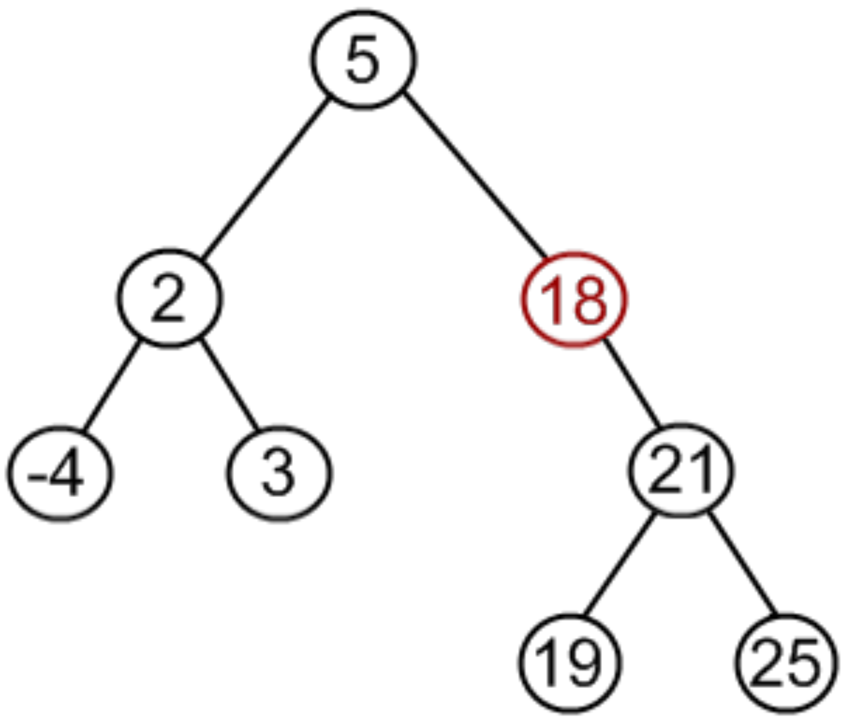
Deletion

- Case 2: Node to be removed has one child.
  - It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

Deletion

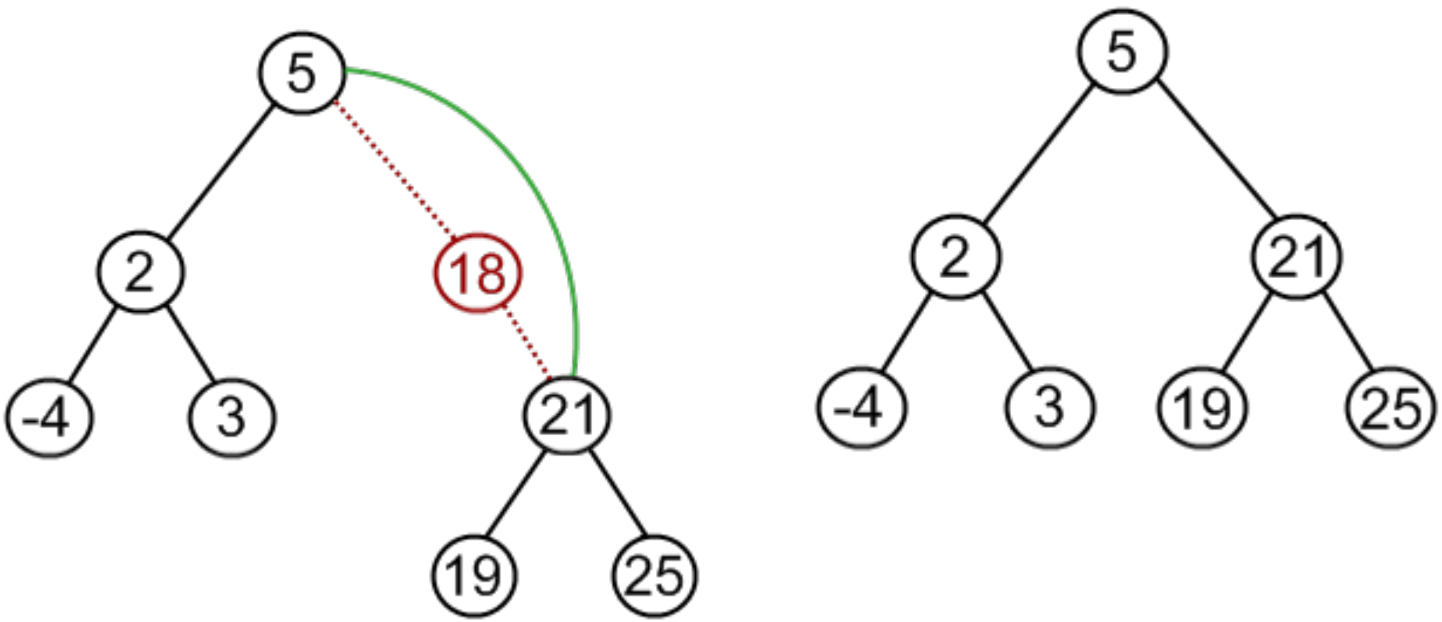
- Case 2: Node to be removed has one child.

  - **Example.** Remove 18 from a BST.



Deletion

- Case 2: Node to be removed has one child.
  - **Example.** Remove 18 from a BST.

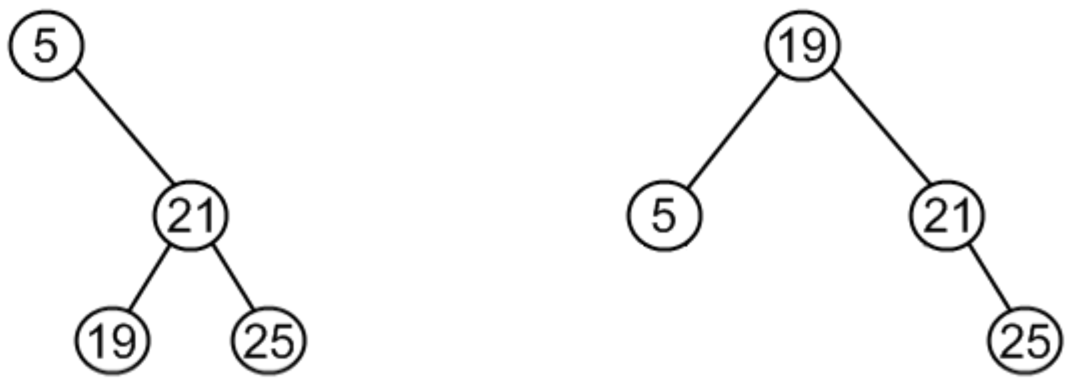


Deletion

- Case 3: Node to be removed has two children.
  - This is the most complex case.
  - To solve it, let us see one useful BST property first.

Deletion

- Case 3: Node to be removed has two children.
  - We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example those BSTs:



Deletion

- Case 3: Node to be removed has two children.
  - The above trees contain the same values {5, 19, 21, 25}.
    To transform first tree into second one, we can do following:
    - choose minimum element from the right subtree (19 in the example);
    - replace 5 by 19;
    - hang 5 as a left child.

Deletion

- The same approach can be utilized to remove a node, which has two children:
  - find a minimum value in the right subtree;
  - replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!
  - apply remove to the right subtree to remove a duplicate.

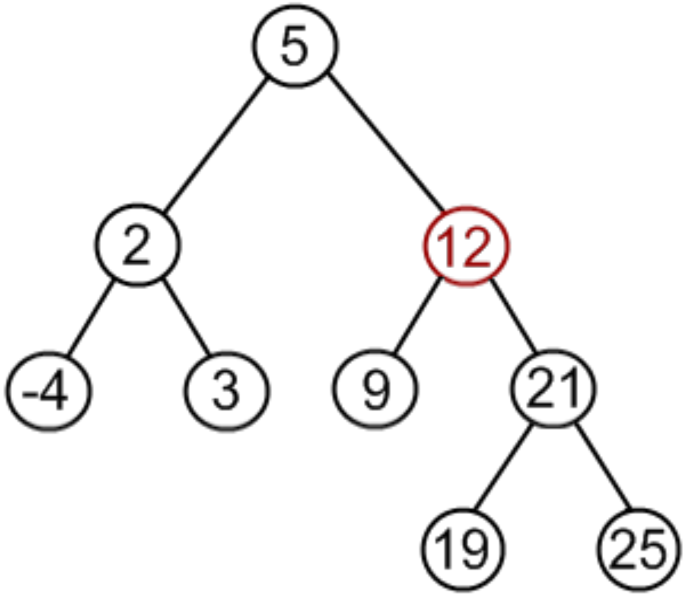
Deletion

- Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

Deletion

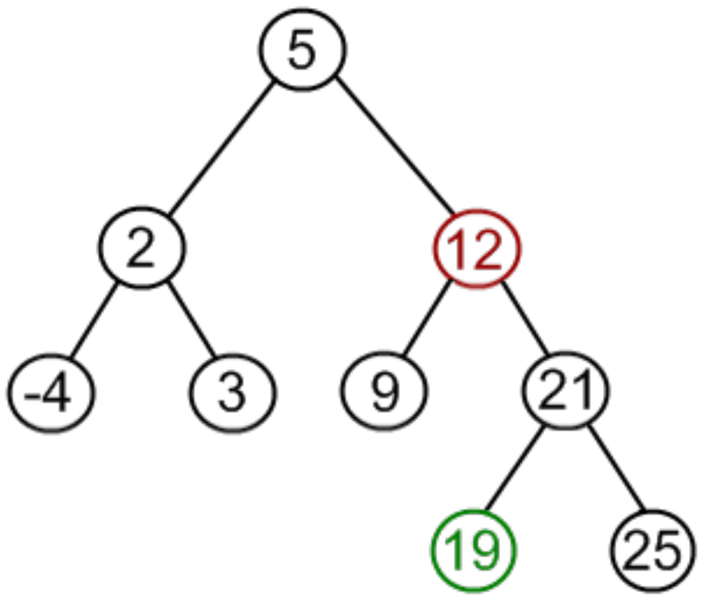
- Case 3: Node to be removed has two children.

  - **Example.** Remove 12 from a BST.



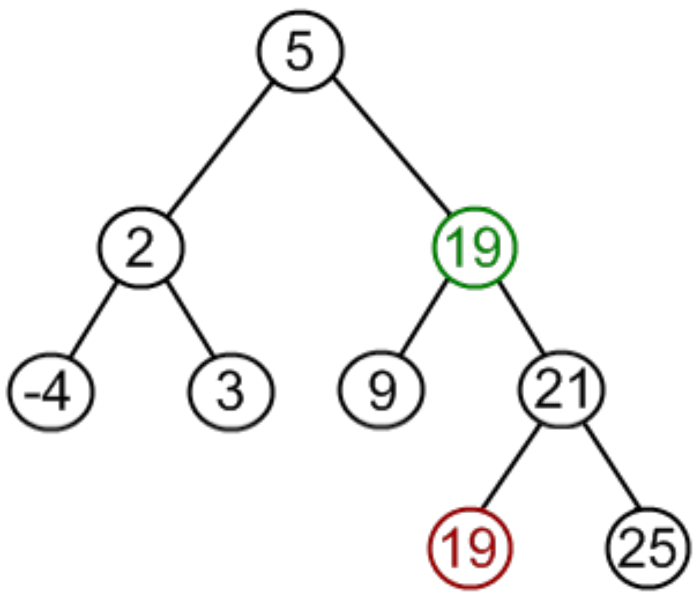
Deletion

- Case 3: Node to be removed has two children.
  - Find minimum element in the right subtree of the node to be removed. In current example it is 19.



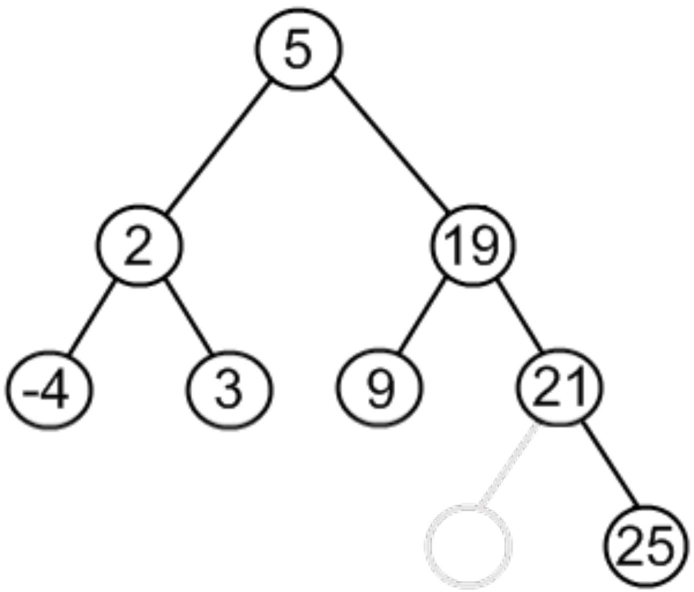
Deletion

- Case 3: Node to be removed has two children.
  - Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.



Deletion

- Case 3: Node to be removed has two children.
  - Remove 19 from the left sub-tree.



```
BSTNode* BSTree::deleteNode(BSTNode*& n,const int v) {
   BSTNode* min, *temp;
   if(n == NULL)
      return NULL;
   else if(v < n->data)
      return deleteNode(n->left,v);
   else if(v > n->data)
      return deleteNode(n->right, v);
   else {
      if(n->left && n->right) {
         min = findMin(n->right);
         n->data = min->data;
         deleteNode(n->right,n->data);
      }
      else {
        temp = n;
        if(n->left == NULL)
           n = n->right;
        else if(n->right == NULL)
           n = n->left;
        delete temp;
      }
      return n;
   }
}
```

BSTree.cpp

Chapter 6 End