ENSF 519: Special Topics in Software Engineering
Data Structures – Trees

Mohammad Moshirpour

Fall 2018

Outline

- Tree
  - Terminology
- Binary Tree
- Binary Search Tree
  - Inserting to a BST

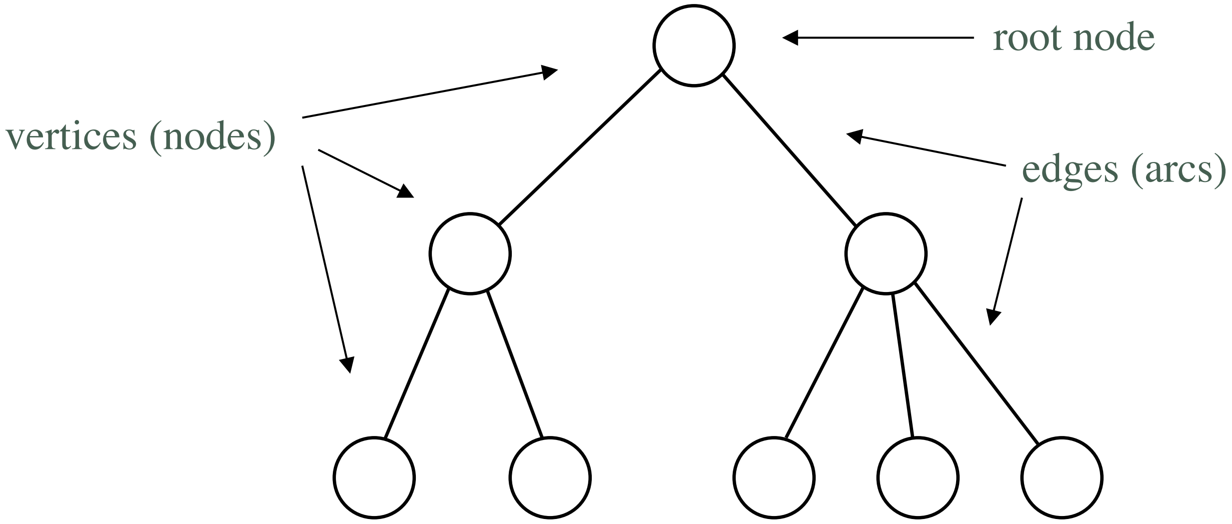
Goal

- In this lecture we will learn about trees, specially Binary Search Tree (BST), which is a hierarchical data structure. We will also take a look at algorithm to insert data in BST.

Introduction

- A tree is a hierarchical data structure
- Is a collection of *vertices* (*nodes*) and *edges* (*arcs*)
  - A vertex contains data and pointer information
  - An edge connects 2 vertices
- Is drawn to grow downwards
  - The *root* node is at the top of the structure
- *Binary tree* has a maximum of two descendants from the node
- *Ternary tree* has a maximum of three descendants from the node

Introduction (cont'd)



Introduction (cont'd)

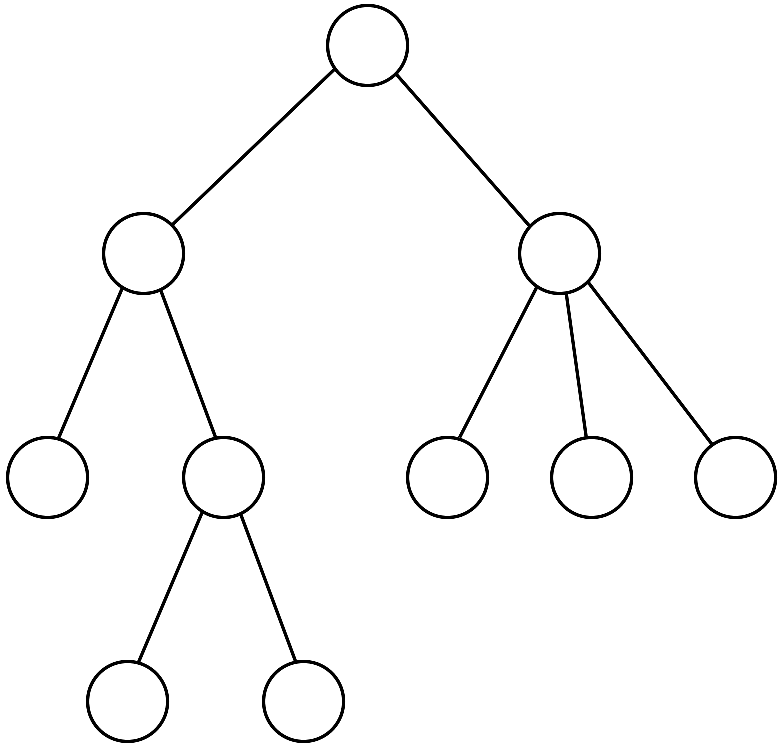
- Each node, except the root, has only one node above it, called the *parent* node
- A node may have zero or more *children*, drawn below it
- Nodes with the same parent are called *twins* or *siblings*
- Nodes with no children are called *leaf* nodes
  - Or *terminal* or *external* nodes

Introduction (cont’d)

- Any node is the root of a *subtree*
  - Consists of it and the nodes below it
- A set of trees is called a *forest*
- A tree consists of levels

Introduction (cont'd)

- E.g.



Level
0
1
2
3

Introduction (cont'd)

- The *height* (*depth*) of a tree is the distance from the root to the node(s) furthest away
  - Is 3 for the above example
- The path length is the sum of edges from each node to the root
  - Is 18 for the above example

Introduction (cont’d)

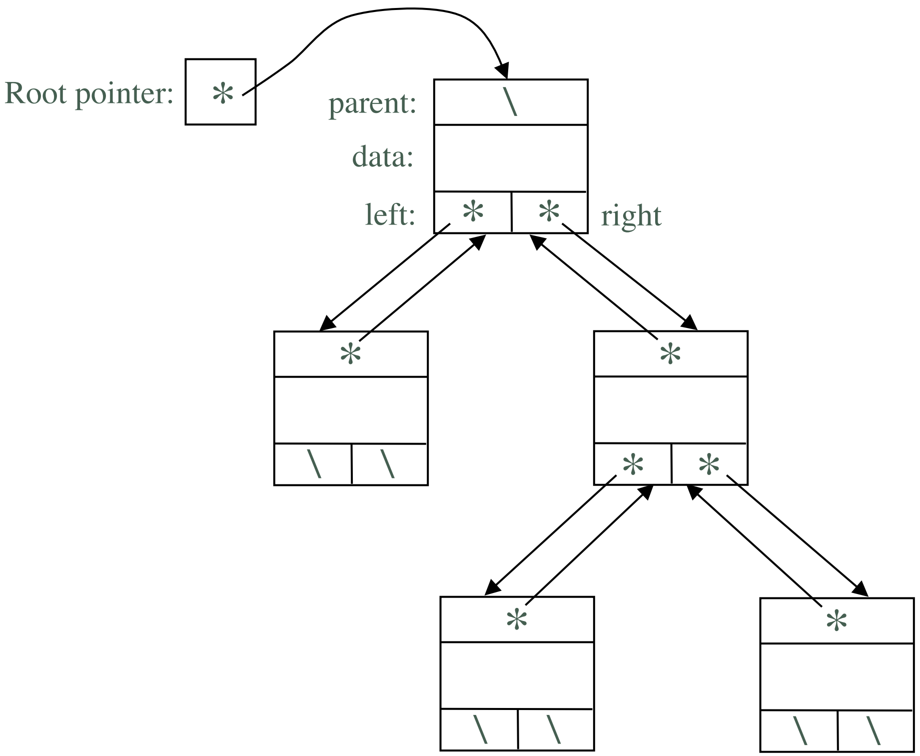
- With *ordered* trees, the order of the children at every node is specified
  - Are much more useful than *unordered* trees

Binary Trees

- Are trees where every node has 0, 1, or 2 children
- Each node contains:
  - Data
  - A left child pointer
  - A right child pointer
  - A parent pointer (optional)
- A root pointer is used to point to the root node

Binary Trees (cont’d)

- E.g.



Binary Trees (cont'd)

- In Java, each node is an object of a class such as the following:

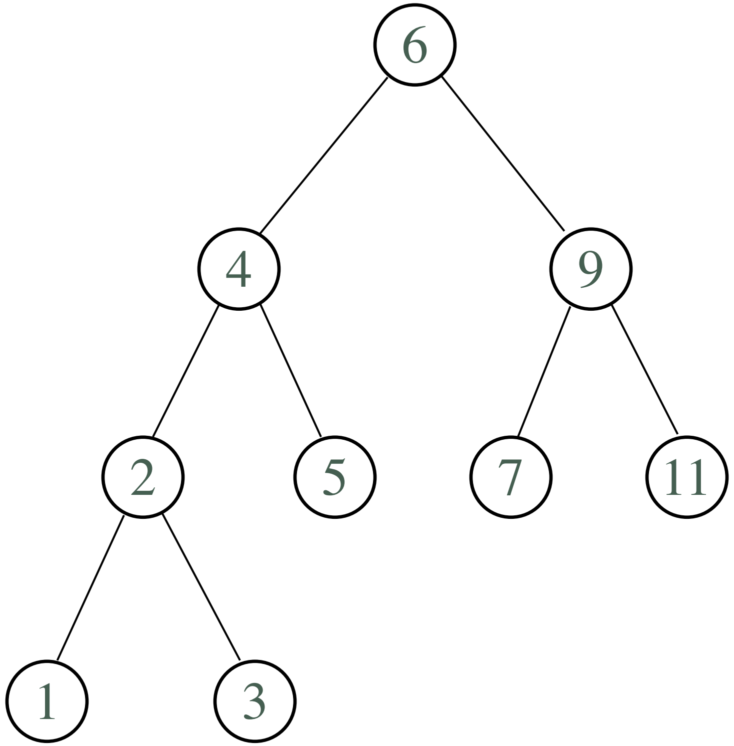
```
public class Node {
  private int data;
  private Node parent, left, right;

  public Node(int el, Node p, Node l, Node r) {
     data = el;
     parent = p;
     left = l;
     right = r;
  }
  . . .
}
```

Binary Search Trees (BST)

- Also called *ordered binary trees*
- Are binary trees organized so that:
  - Every left child is less than (or equal to) the parent node
  - Every right child is greater than the parent node
- E.g.

Binary Search Trees (cont'd)

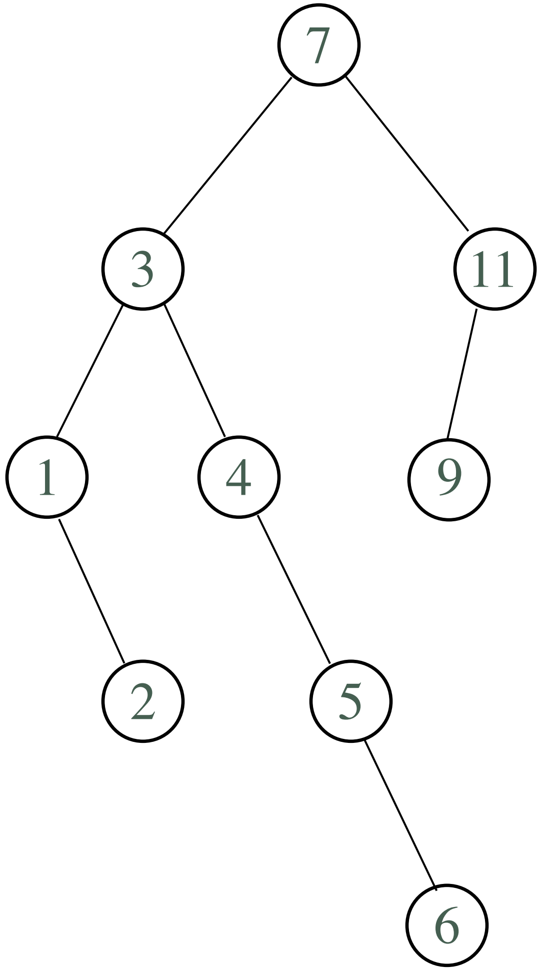
- E.g.



Binary Search Trees (cont'd)

- All nodes in any left subtree will be less than (or equal to) the parent node
- All nodes in any right subtree will be greater than the parent node
- Different binary search trees can represent the same data
  - The shape of the tree depends on the order of insertion
  - E.g.

Binary Search Trees (cont’d)



Binary Search Trees (cont'd)

- In the best case, the tree is balanced and the height is minimized
  - Height is approximately $\lg(n)$
- In the worst case, the tree degenerates into a linked list
  - Height is $n-1$

Binary Search Trees (cont’d)

- If the tree is well balanced, searches are efficient
  - $O(\lg n)$
  - Related to the binary search of an sorted array

Binary Search Trees (cont'd)

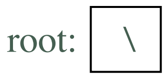
- Insertion:
  - Requires a search of the existing tree, to find the parent node of the new node
    - New nodes are always added as leaf nodes
  - The new node is then attached to the parent
  - If the tree is empty, then the new node becomes the root node
  - Iterative implementation:

Binary Search Trees (cont'd)

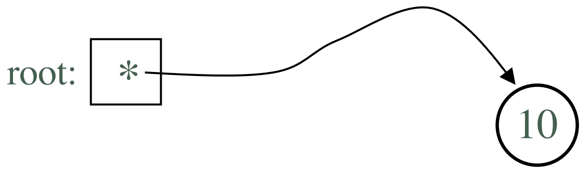
```
public void insert(int el, Node root)
{
   Node current = root, parent = null;

   while (current != null) {
      parent = current;
      if (el > current.data)
         current = current.right;
      else
         current = current.left;
   }

   if (root == null)
      root = new Node(el, parent, null, null);
   else if (el > parent.data)
      parent.right = new Node(el, parent, null, null);
   else
      parent.left = new Node(el, parent, null, null);
}
```

Binary Search Trees (cont'd)

- E.g. Successively insert 10, 5, 17, 12, 20, 2
  - Initial state:

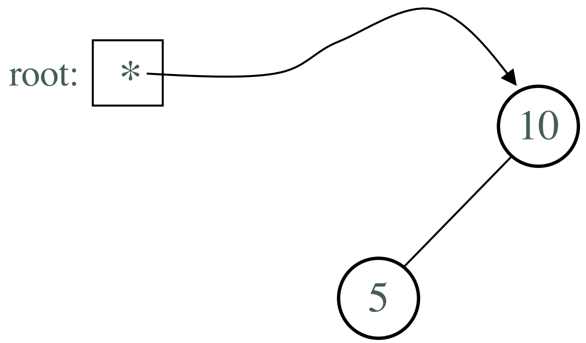


– Insert 10:



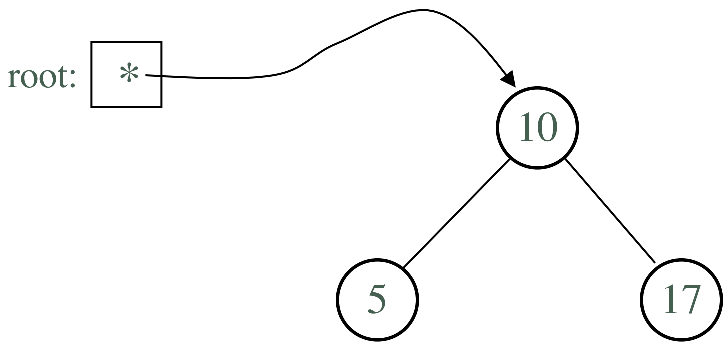
Binary Search Trees (cont'd)

- Insert 5:



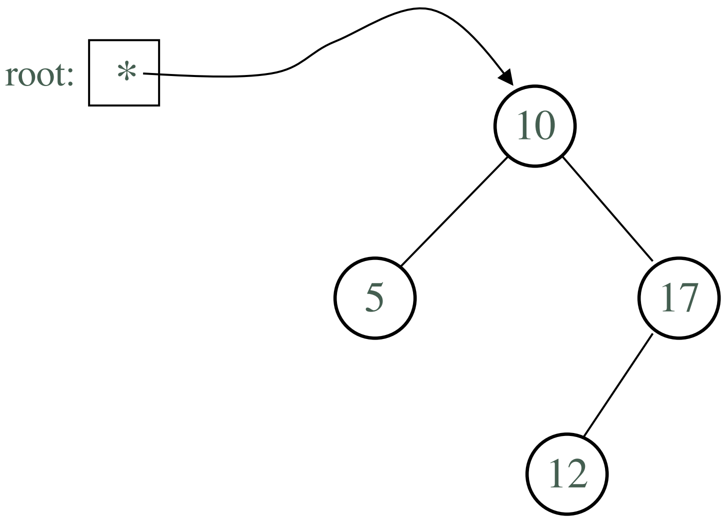
Binary Search Trees (cont'd)

- Insert 17:



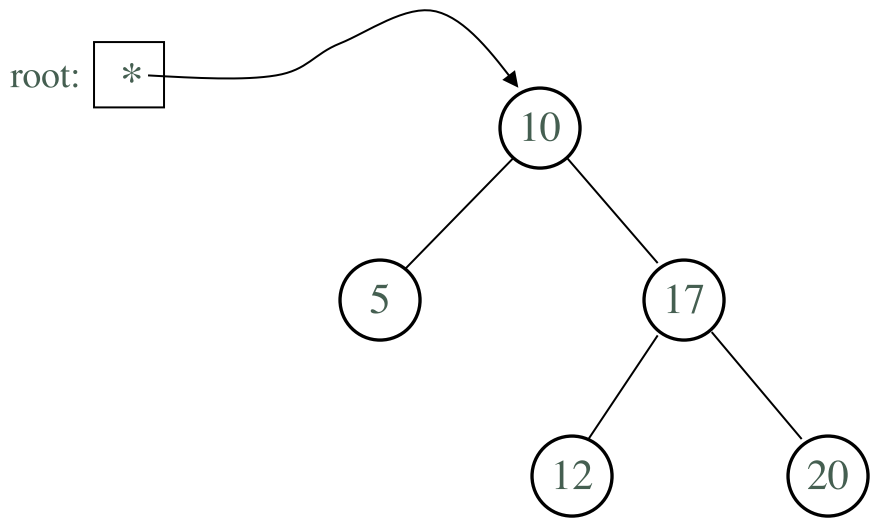
Binary Search Trees (cont'd)

- Insert 12:



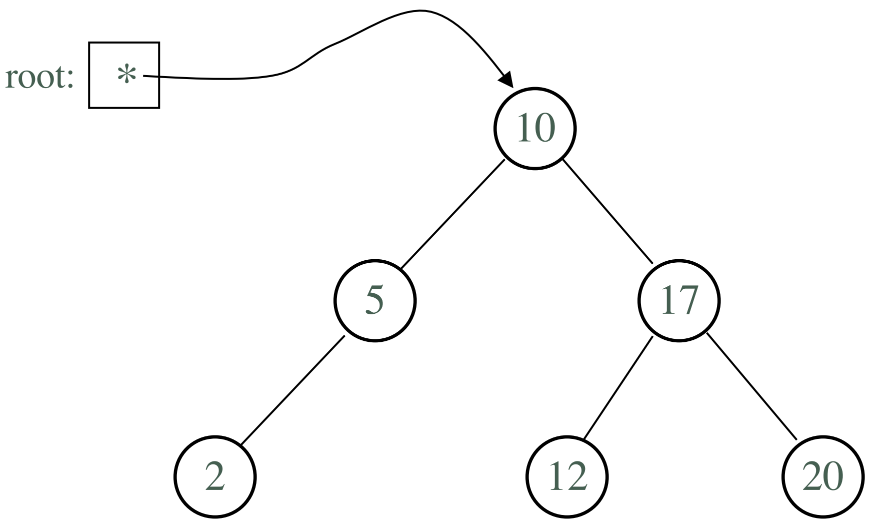
Binary Search Trees (cont'd)

- Insert 20:



Binary Search Trees (cont'd)

- Insert 2:



Binary Search Trees (cont’d)

- Traversal
  - To traverse a tree, all nodes are *visited* once in some prescribed order
  - Two types:
    - *Depth-first*: recursively visit each node starting at the far left (or right)
    - *Breadth-first*: starting at the highest level, move down level by level, visiting nodes on each level from left to right
      - Can also start at the bottom, or traverse from right to left

Binary Search Trees (cont'd)

- *Depth-first, in-order* traversal
  - Visits nodes in ascending order
  - Implementation:

```
public void inorder(Node current)
{
   if (current != null) {
      inorder(current.left);
      // visit current node; for example:
      System.out.println(current.toString());
      inorder(current.right);
   }
}
```

Binary Search Trees (cont'd)

- Would be called from client code as follows:

```
Node root = null;
// Build tree doing successive insertion
. . .

// Traverse tree
inorder(root);

. . .
```

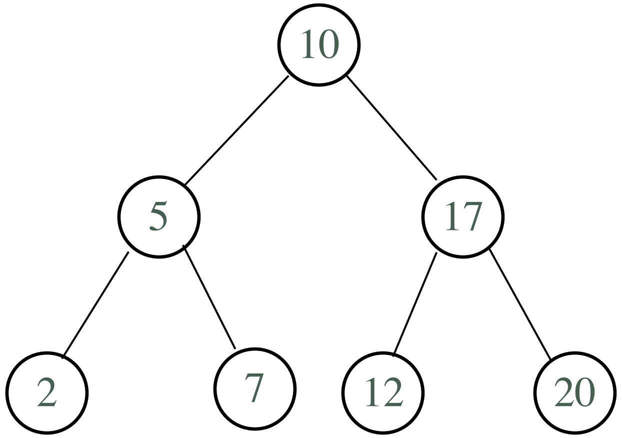
Binary Search Trees (cont'd)

- *Depth-first, pre-order* traversal
  - Processes the node first, then the left subtree, then the right subtree
  - Implementation:

```
public void preorder(Node current)
{
   if (current != null) {
      // visit current node; for example:
System.out.println(current.toString());
      preorder(current.left);
      preorder(current.right);
   }
}
```

Binary Search Trees (cont'd)

- The tree:



would be processed as: 10, 5, 2, 7, 17, 12, 20

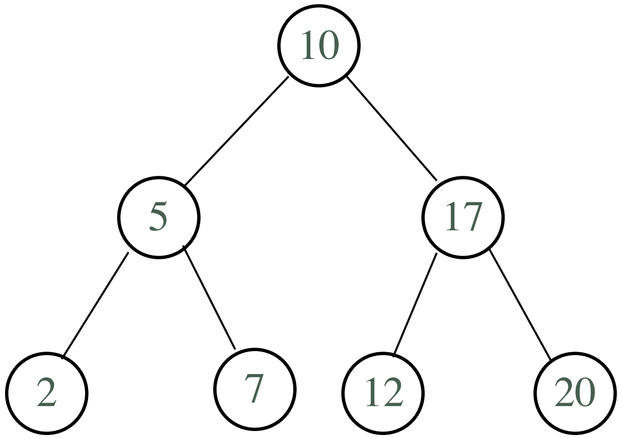
Binary Search Trees (cont'd)

- *Depth-first, post-order* traversal
  - Processes the left subtree, then the right subtree, then the node
  - Implementation:

```
public void postorder(Node current)
{
   if (current != null) {
      postorder(current.left);
      postorder(current.right);
      // visit current node; for example:

System.out.println(current.toString());
   }
}
```

Binary Search Trees (cont’d)

- The tree:



would be processed as: 2, 7, 5, 12, 20, 17, 10

Binary Search Trees (cont’d)

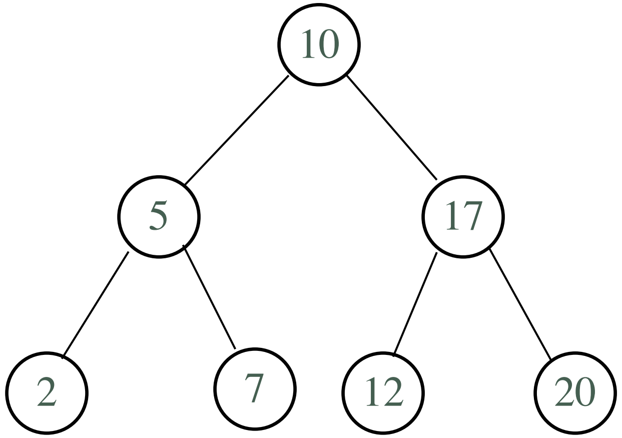
- The depth-first traversals could be implemented non-recursively
  - Requires iteration and an explicit stack
  - Less elegant than the recursive implementation

Binary Search Trees (cont'd)

- Breadth-first traversal
  - Requires use of a queue
  - Top-down, left-to-right implementation (*Drozdek* p. 224):

Binary Search Trees (cont'd)

- The tree:



would be processed as: 10, 5, 17, 2, 7, 12, 20

Binary Search Trees (cont'd)

- Searching
  - Can be done iteratively:

```
public Node search(Node current, int key)
{
   while (current != null) {
      if (key == current.data)
         return current;  // found
      else if (key < current.data)
         current = current.left;
      else
         current = current.right;
   }
   return null;  // not found
}
```

Binary Search Trees (cont’d)

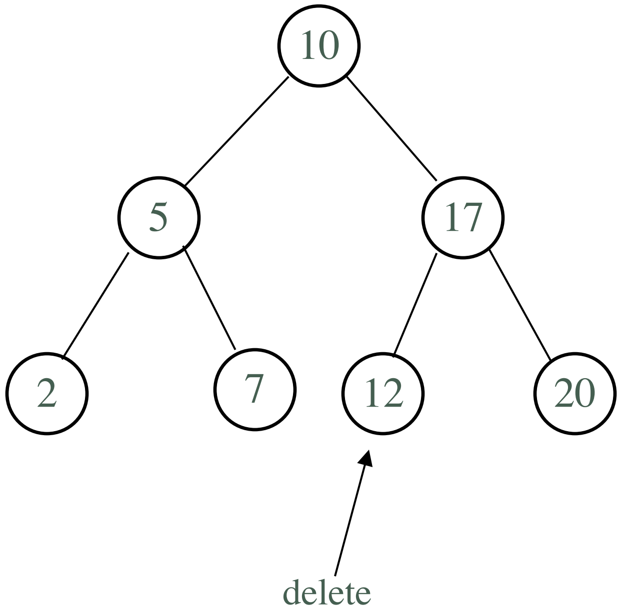
- Is very efficient when performed on a “well-balanced” tree
  - Is $O(\lg n)$ in when the height of the tree is minimized
  - Is also $O(\lg n)$ when the tree is formed by inserting nodes in random input order
- Is less efficient if the tree has degenerated to a linked list
  - Is $O(n)$

Binary Search Trees (cont’d)

- Deleting nodes
  - 3 cases:
    - Deleting a leaf node
      - Set the parent node’s child pointer to null
      - Free the deleted node’s memory

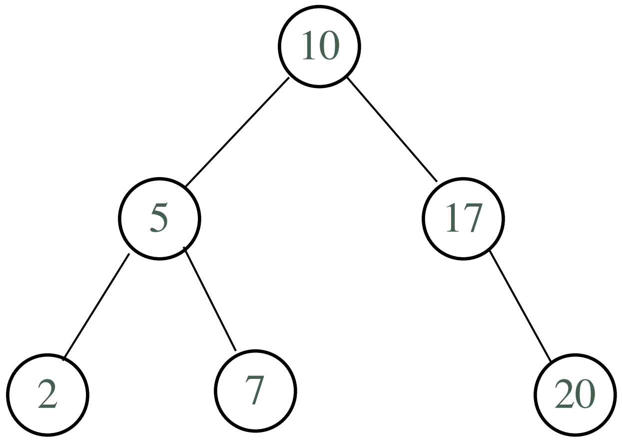
Binary Search Trees (cont'd)

- E.g. Deleting 12 from the tree:



Binary Search Trees (cont’d)

Results in:

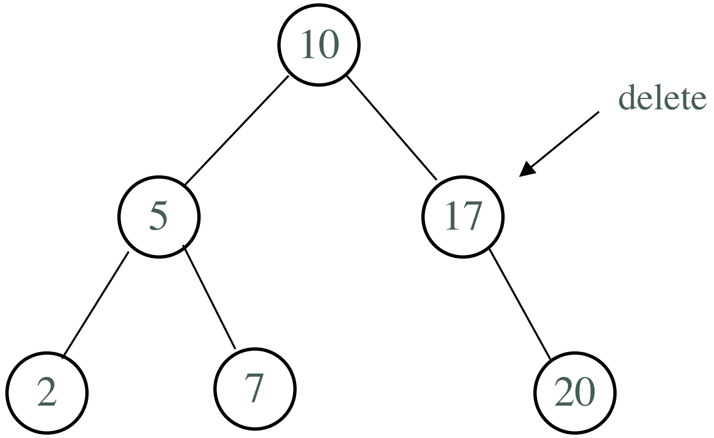


Binary Search Trees (cont’d)

- Deleting a node with only one child
  - Set the parent node’s child pointer to the child of the deleted node (“splice out” the node)
  - Free the deleted node’s memory

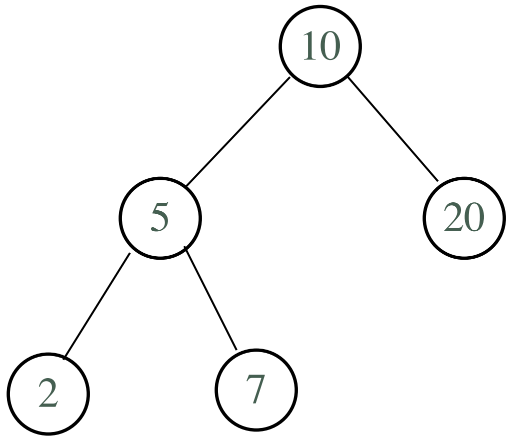
Binary Search Trees (cont'd)

- E.g. Deleting 17 from the tree:



Binary Search Trees (cont'd)

Results in:

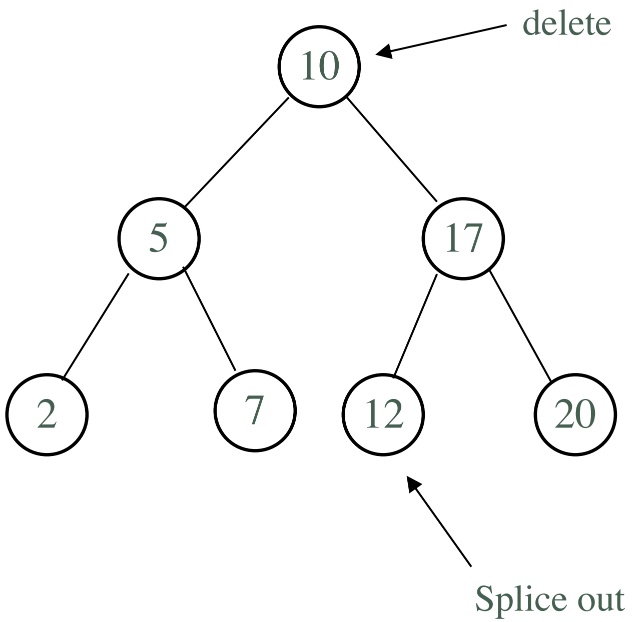


Binary Search Trees (cont’d)

- Deleting a node with two children
  - Find the *smallest* node in the *right* subtree below the node to delete
  - “Splice out” that node, using the steps from one of the cases above
  - Substitute the spliced node for the deleted node, either by copying, or by adjusting pointers
  - Free the deleted node’s memory
  - Note: could use the *largest* node in the *left* subtree for first step above

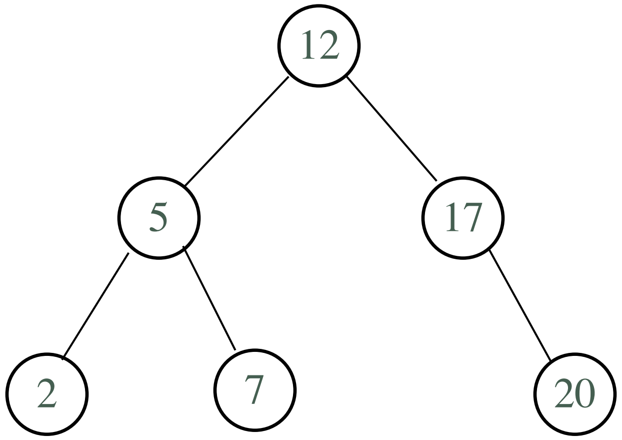
Binary Search Trees (cont'd)

- E.g. Deleting 10 from the tree:



Binary Search Trees (cont'd)

Results in:



Summary

- A tree is a hierarchical data structure that is a collection of vertices (nodes) and edges (arcs)
- Binary trees Are trees where every node has 0, 1, or 2 children
- Are binary search trees organized so that:
  - Every left child is less than (or equal to) the parent node
  - Every right child is greater than the parent node
- Insertion:
  - Requires a search of the existing tree, to find the parent node of the new node
    - New nodes are always added as leaf nodes
  - The new node is then attached to the parent
  - If the tree is empty, then the new node becomes the root node

Summary (Cont’d)

- Traversal:
  - All nodes are visited once in some prescribed order.
  - Has two types:
    - Depth-first
    - Breadth-first
- Searching:
  - Is very efficient when performed on a “well-balanced” tree.
- Deleting:
  - A leaf node
  - A node with only one child
  - A node with two children

Review Questions

- What is a tree?
- What is a binary tree?
- What is a binary search tree?
- Define these terms (in tree terminology):
  - Edge
  - Vertex
  - Parent
  - Child
  - Sibling
  - Root
  - Leaf
  - Height (depth)
  - Path length

Review Questions (Cont’d)

- What is the content of each node in a binary tree?
- What is the approximate height of a binary search tree with n nodes?
- How does the binary tree insertion work?
- What are the type of binary tree traversal?
- How does Depth-first work?
- How does Breadth-first work?
- When is the BTS efficient?
- How does the binary tree deleting work?

Any questions?