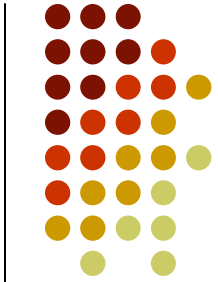
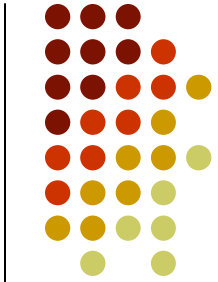
Deletion algorithm – Phase 2: Remove node or replace its with successor



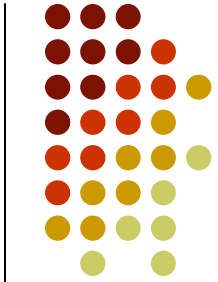
```
TreeNode<T> deleteNode(TreeNode<T> n) {
  // Returns a reference to a node which replaced n.
    // Algorithm note: There are four cases to consider:
    //   1. The n is a leaf.
    //   2. The n has no left child.
    //   3. The n has no right child.
    //   4. The n has two children.
    // Calls: findLeftmost and deleteLeftmost

    // test for a leaf
    if (n.getLeft() == null && n.getRight() == null)
      return null;

    // test for no left child
    if (n.getLeft() == null)
      return n.getRight();

    // test for no right child
    if (n.getRight() == null)
      return n.getLeft();

    // there are two children: retrieve and delete the inorder successor
    T replacementItem = findLeftMost(n.getRight()).getItem();
    n.setItem(replacementItem);
    n.setRight(deleteLeftMost(n.getRight()));
    return n;
  }  // end deleteNode
```

Deletion algorithm – Phase 3: Remove successor



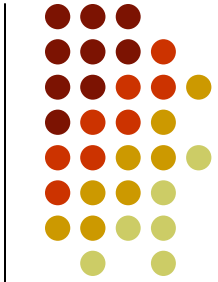
```
TreeNode<T> findLeftmost(TreeNode<T> n)  {
    if (n.getLeft() == null) {
      return n;
    }
    else {
      return findLeftmost(n.getLeft());
    }  // end if
  }  // end findLeftmost

  TreeNode<T> deleteLeftmost(TreeNode<T> n)
  // Returns a new root.
  {
    if (n.getLeft() == null) {
      return n.getRight();
    }
    else {
      n.setLeft(deleteLeftmost(n.getLeft()));
      return n;
    }  // end if
  }  // end deleteLeftmost
```

BST Efficiency

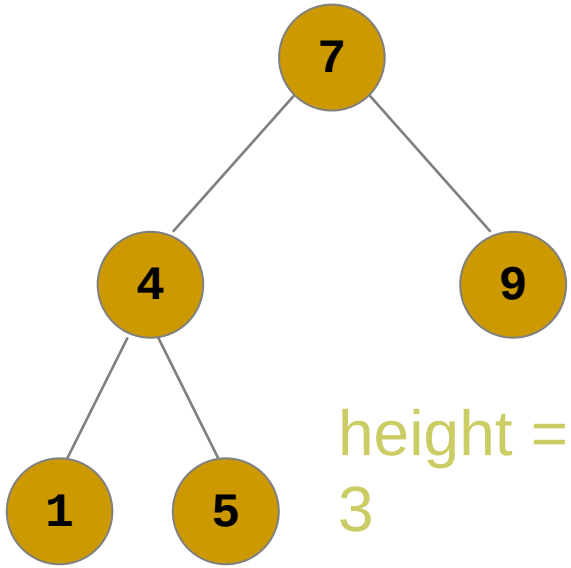


- The efficiency of BST operations depends on the height of the tree
- All three operations (search, insert and delete) are O(*height*)
- If the tree is complete/full the height is $\lfloor \log(n) \rfloor + 1$
- What if it isn't complete/full?

Height of a BST



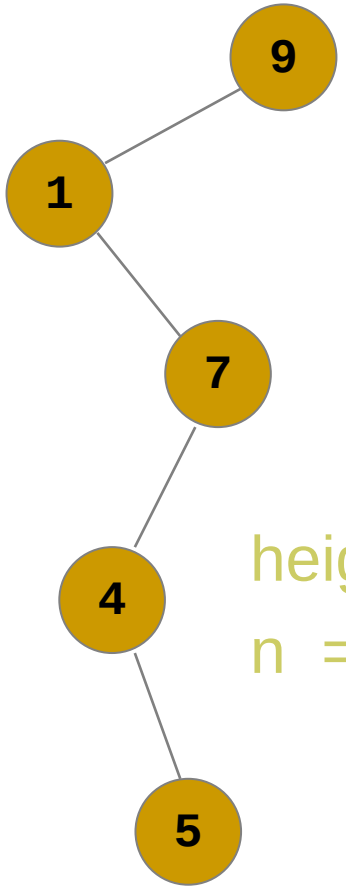
- Insert 7
- Insert 4
- Insert 1
- Insert 9
- Insert 5
- It’s a complete tree!



height = $\lfloor \log(5) \rfloor + 1 = 3$

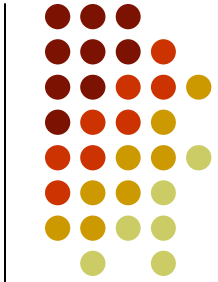
Height of a BST

- Insert 9
- Insert 1
- Insert 7
- Insert 4
- Insert 5
- It’s a linked list!

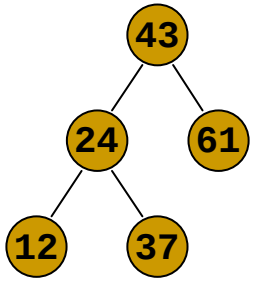


height =
n = 5 = O(n)

Binary Search Trees – Performance

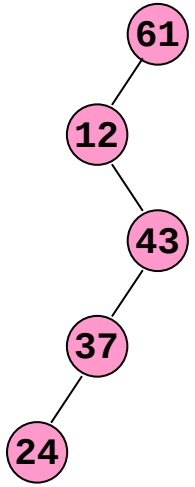


- Items can be inserted in and removed and removed from BSTs in O(*height*) time
- So what is the height of a BST?
  - If the tree is complete it is O(log *n*) [best case]
  - If the tree is not balanced it may be O(*n*) [worst case]



complete BST

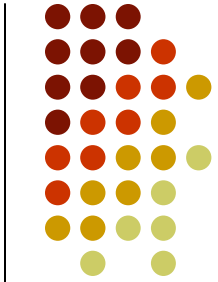
height = O(log*n*)



incomplete BST

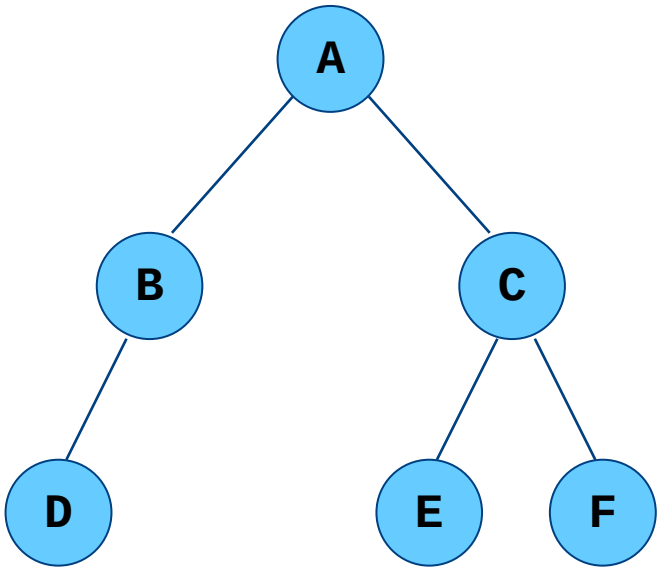
height = O(*n*)

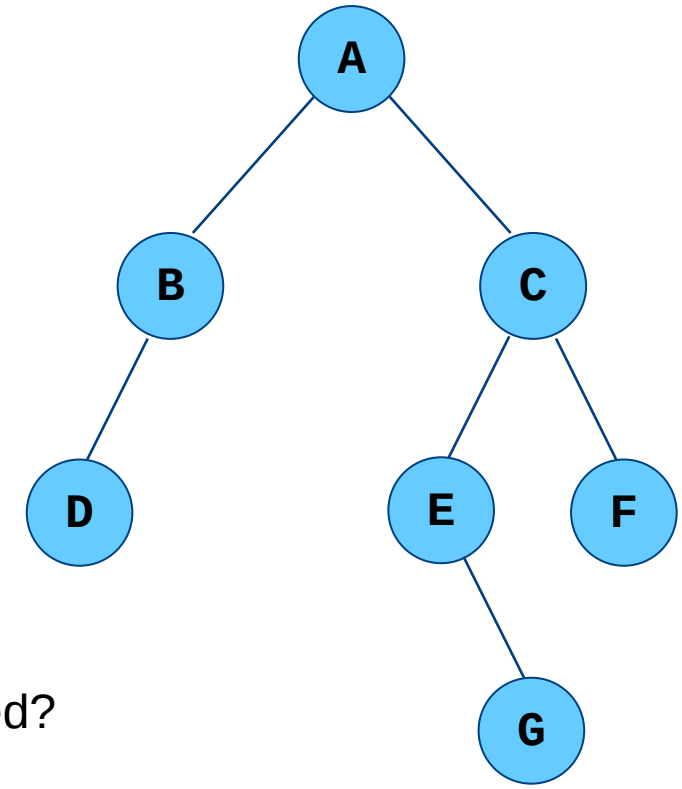
BSTs with heights O(log n)



- It would be ideal if a BST was always close to a full binary tree
  - It's enough to guarantee that the height of tree is *O(*log *n)*
- To guarantee that we have to make the structure of the tree and insertion and deletion algorithms more complex
  - e.g. AVL trees (balanced), 2-3 trees, 2-3-4 trees (full but not binary), **`red-black`** trees (if red vertices are ignored then it's like a full tree)

Example:
Balanced Binary Trees



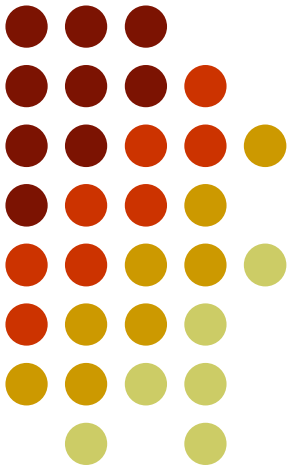


Question: Is it enough to assume balanced?

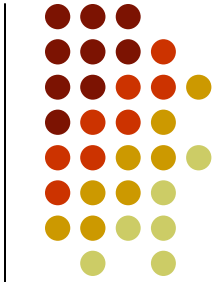
Yes, it's enough. The height of a balanced tree is at most 2.log *n*

CMPT 225

Red–black trees

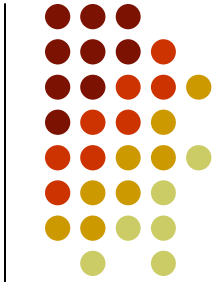


Red-black Tree Structure



- A red-black tree is a **BST**!
- Each node in a red-black tree has an extra color field which is
  - **red** or
  - **black**
- In addition **false** nodes are added so that every (real) node has two children
  - These are **pretend** nodes, they don't have to have space allocated to them
  - These nodes are colored black
  - We do not count them when measuring a height of nodes
- Nodes have an extra reference to their parent

Red-black Tree Properties



- 1 – Every node is either red or black
- 2 – Every leaf is black
  - This refers to the **pretend** leaves
  - In implementation terms, every null child of a node considered to be a black leaf
- 3 – If a node is **red** both its children must be black
- 4 – Every path from a node to its descendent leaves contains the same number of black nodes
- 5 – The root is black (mainly for convenience)