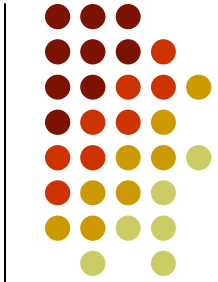
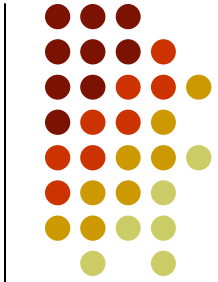
# InOrder Traversal Algorithm



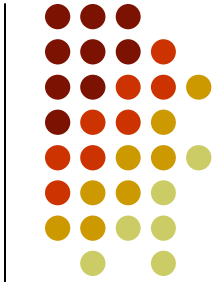
```
// InOrder traversal algorithm
inOrder(TreeNode<T> n) {
    if (n != null) {
        inOrder(n.getLeft());
        visit(n)
        inOrder(n.getRight());
    }
}
```

# Examples



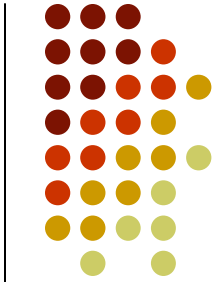
- Iterative version of in-order traversal
  - Option 1: using Stack
  - Option 2: with references to parents in TreeNodes
- Iterative version of height() method

# Binary Tree Implementation



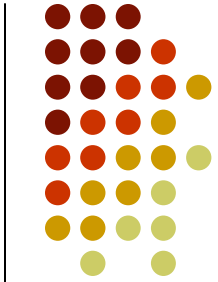
- The binary tree ADT can be implemented using a number of data structures
  - Reference structures (similar to linked lists), as we have seen
  - Arrays – either simulating references or complete binary trees allow for a special very memory efficient array representation (called heaps)
- We will look at 3 applications of binary trees
  - Binary search trees (references)
  - Red-black trees (references)
  - Heaps (arrays)

# Problem: Design a data structure for storing data with keys



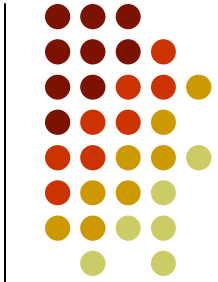
- Consider maintaining data in some manner (data structure)
  - The data is to be frequently searched on the **search key** e.g. a dictionary, records in database
- Possible solutions might be:
  - A sorted array (by the keys)
    - Access in O(log *n*) using binary search
    - Insertion and deletion in linear time
  - An sorted linked list
    - Access, insertion and deletion in linear time

# Dictionary Operations



- The data structure should be able to perform all these operations efficiently
  - Create an empty dictionary
  - Insert
  - Delete
  - Look up (by the key)
- The insert, delete and look up operations should be performed in O(log *n*) time
- Is it possible?

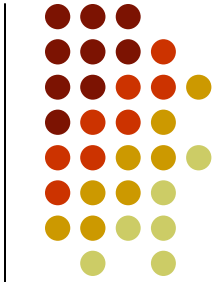
# Data with keys



- For simplicity we will assume that keys are of type `long`, i.e., they can be compared with operators <, >, <=, ==, etc.
- All items stored in a container will be derived from KeyedItem.

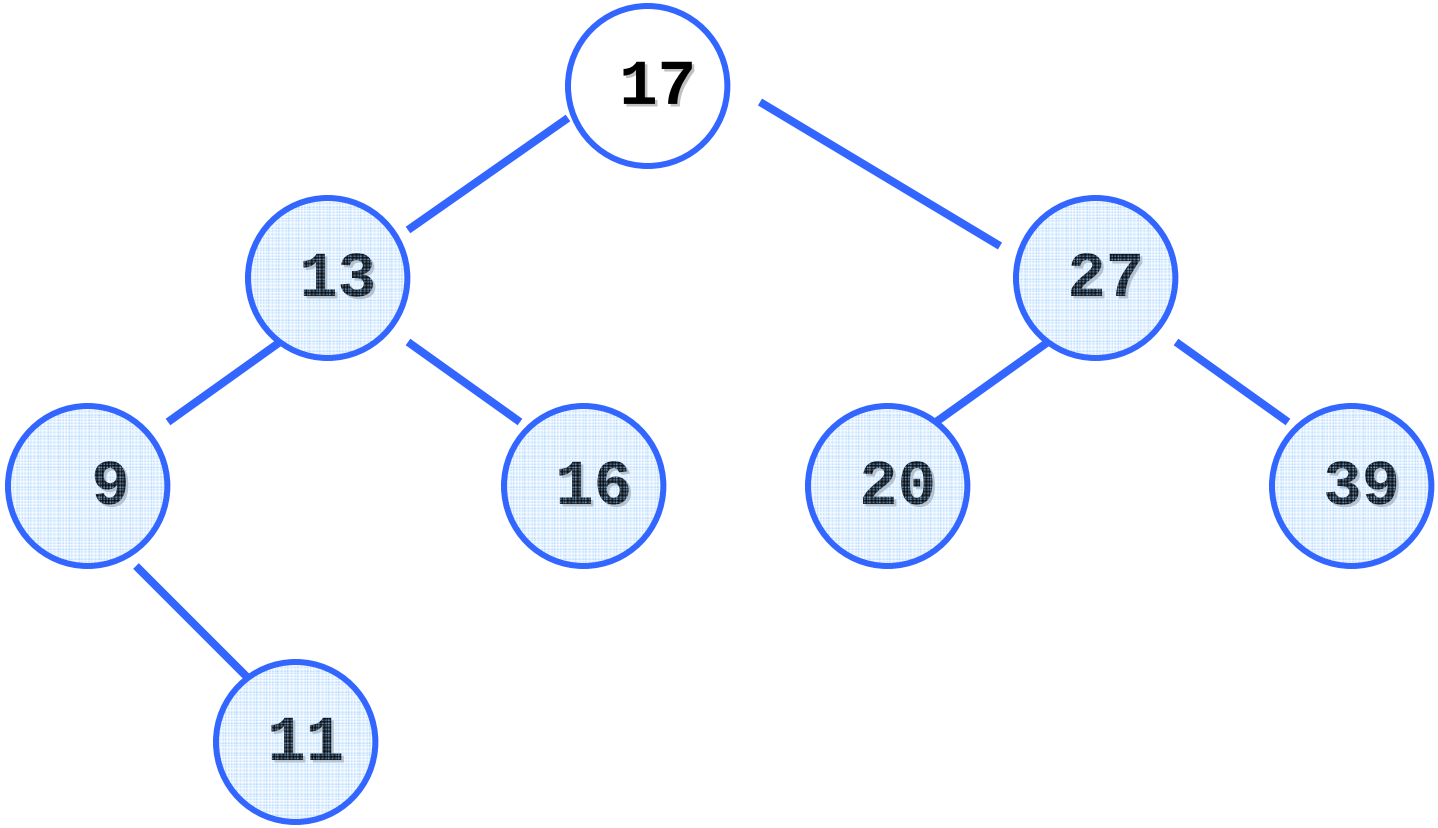
```
public class KeyedItem
{
   private long key;

   public KeyedItem(long k)
   {
        key=k;
   }
   public getKey() {
        return key;
   }
}
```

# Binary Search Trees (BSTs)



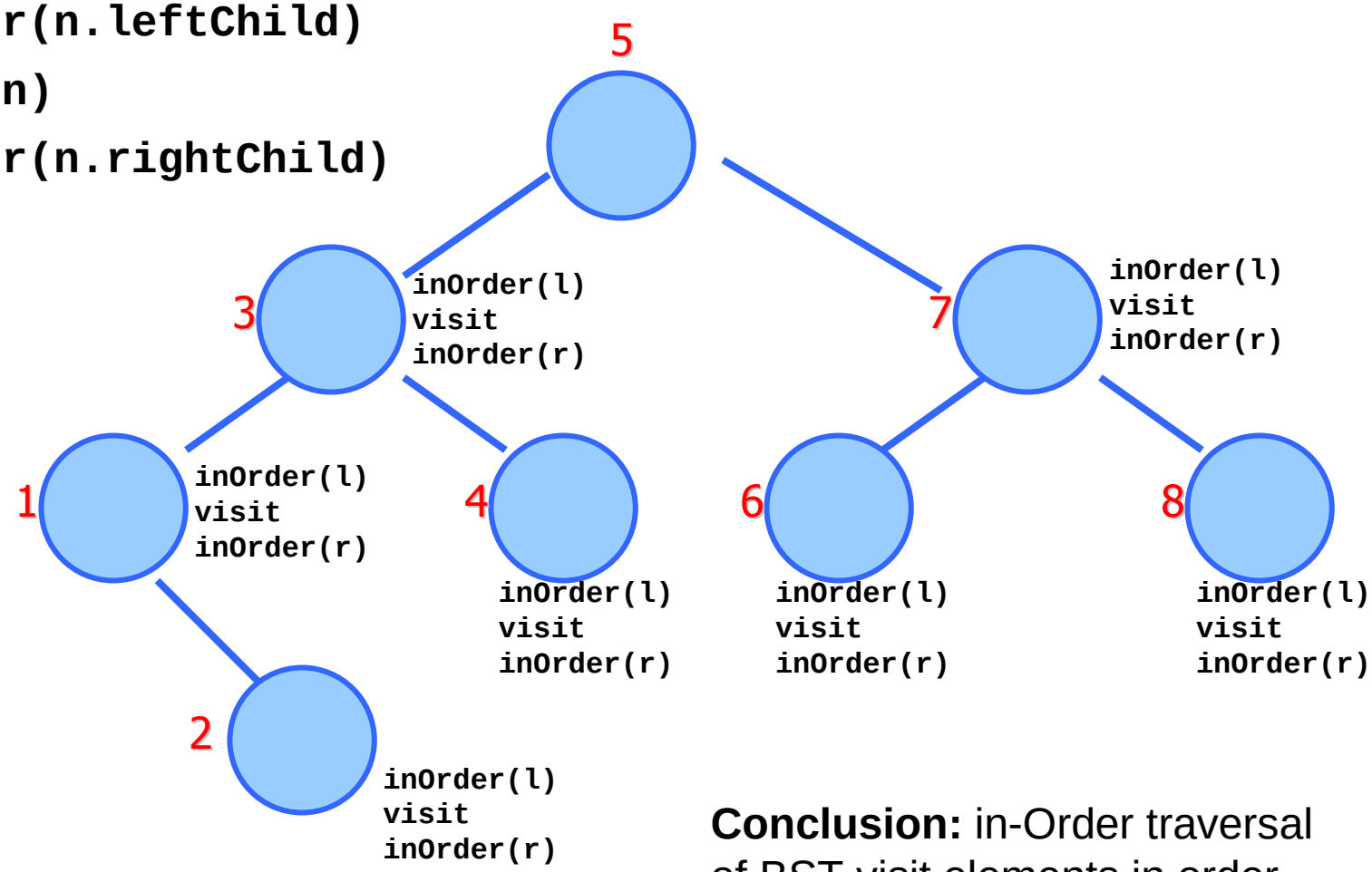
- A binary search tree is a binary tree with a special property
  - For all nodes *v* in the tree:
    - All the nodes in the **left subtree** of *v* contain items **less than** the item in *v* and
    - All the nodes in the **right subtree** of *v* contain items **greater than or equal to** the item in *v*

# BST Example



# BST InOrder Traversal

```
inOrder(n.leftChild)
visit(n)
inOrder(n.rightChild)
```




**Conclusion:** in-Order traversal of BST visit elements in order.