Data Structure

Lecture#11: Binary Trees (Chapter 5)

U Kang
Seoul National University

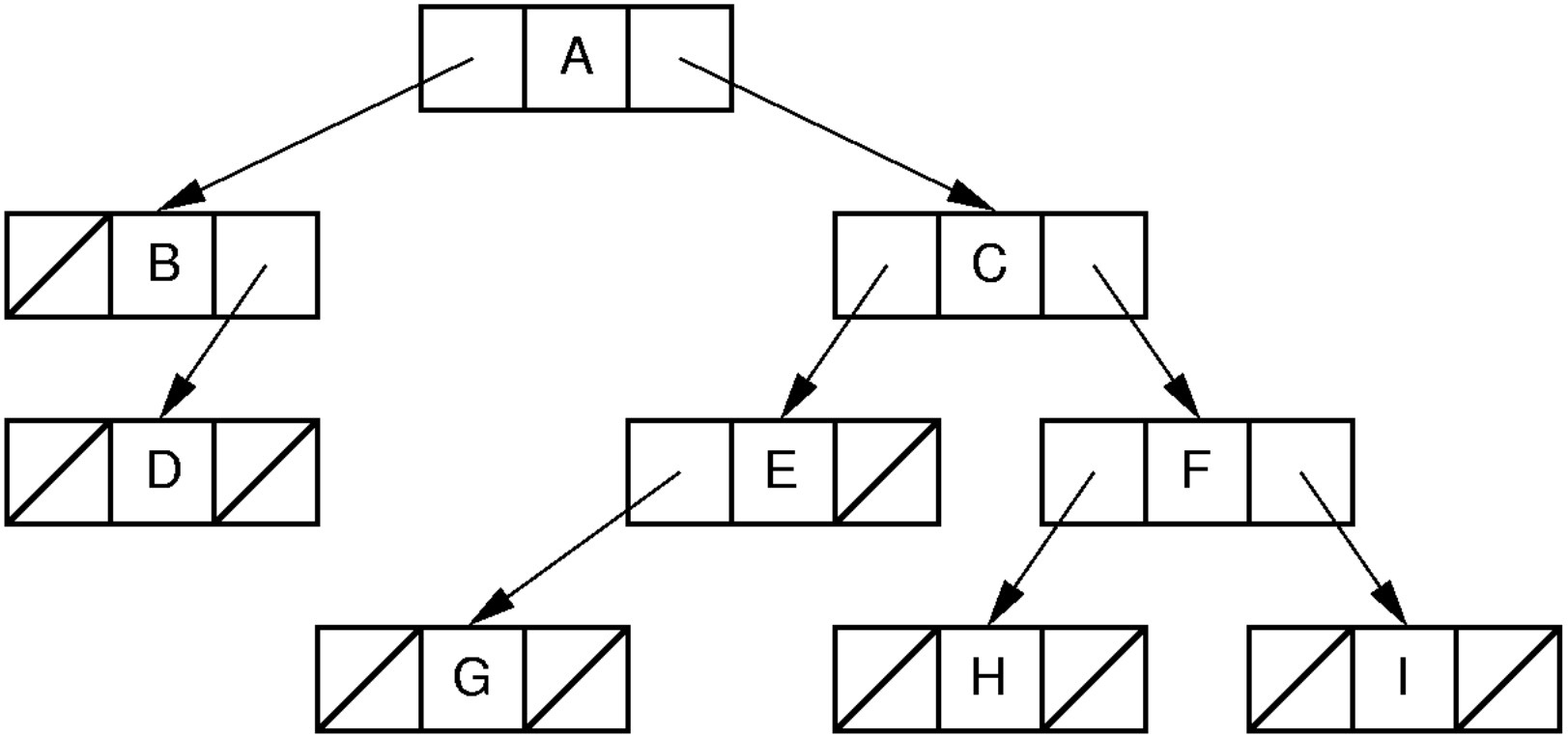
In This Lecture

- Implementation and space overhead of binary tree
- Main idea and operations for BST (Binary Search Tree)
- Complexity and implementations for BST

Recursion Examples

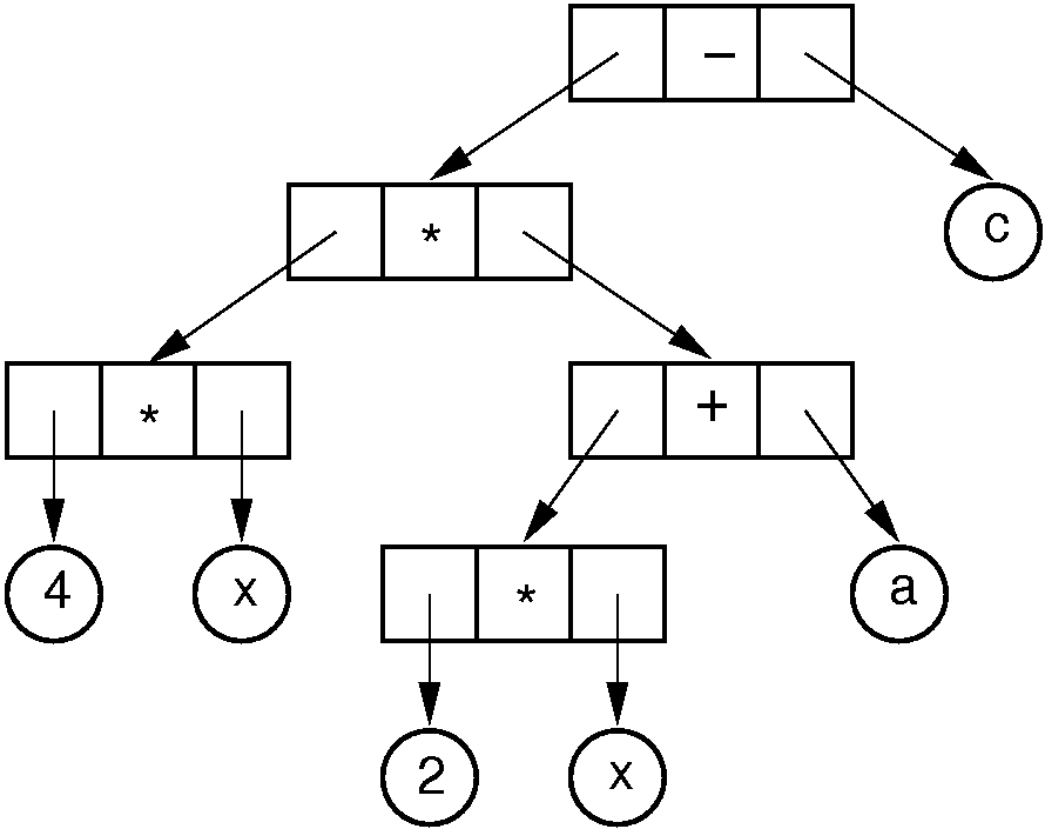
```
int count(BinNode rt) {
  if (rt == null) return 0;
  return 1 + count(rt.left()) +
             count(rt.right());
}
```

Binary Tree Implementation (1)



Leaf implementation is identical to internal node implementation

Binary Tree Implementation (2)



Distinct internal and leaf node implementations

Between implementations (1) and (2), which is better? Why?

Inheritance (1)

```
/** Base class */
public interface VarBinNode {
  public boolean isLeaf();
}

/** Leaf node */
class VarLeafNode implements VarBinNode {
  private String operand;
  public VarLeafNode(String val)
    { operand = val; }
  public boolean isLeaf() { return true; }
  public String value() { return operand; }
};
```

Inheritance (2)

```
/** Internal node */
class VarIntlNode implements VarBinNode {
  private VarBinNode left;
  private VarBinNode right;
  private Character operator;

  public VarIntlNode(Character op,
                     VarBinNode l, VarBinNode r)
    { operator = op; left = l; right = r; }
  public boolean isLeaf() { return false; }
  public VarBinNode leftchild() { return left; }
  public VarBinNode rightchild(){ return right; }
  public Character value() { return operator; }
}
```

Inheritance (3)

```
/** Preorder traversal */
public static void traverse(VarBinNode rt) {
  if (rt == null) return;
  if (rt.isLeaf())
    VisitLeafNode(((VarLeafNode)rt).value());
  else {
    VisitInternalNode(((VarIntlNode)rt).value());
    traverse(((VarIntlNode)rt).leftchild());
    traverse(((VarIntlNode)rt).rightchild());
  }
}
```

Space Overhead (1)

- Space overhead = (non data space) / (total space)
- From the Full Binary Tree Theorem:
  - Half of the pointers are **null**.
- If only leaves store data, then overhead depends on whether the tree is full.
- Ex: Full tree, all nodes the same, with two pointers to children and one to element:
  - Total space required is $(3p + d)n$
  - Overhead: $3pn$
  - If $p = d$, this means $3p/(3p + d) = 3/4$ overhead.

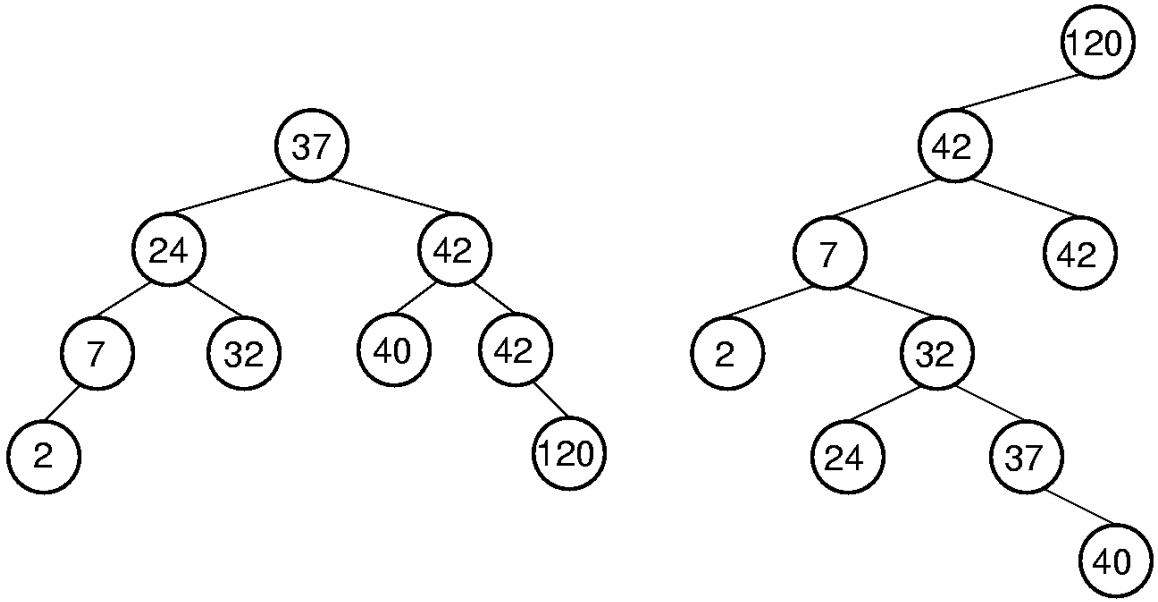
p: space for a pointer
d: space for a data item

Space Overhead (2)

- How to decrease space overhead?
- Idea: eliminate child pointers from the leaf nodes
- Then, space overhead = $\frac{\left(\frac{n}{2}\right)(2p)+np}{\left(\frac{n}{2}\right)(2p)+np+nd} = \frac{2p}{2p+d}$
- If p = d, then the space overhead = 2/3

Binary Search Trees

- BST Property: All elements stored in the left subtree of a node with value *K* have values < *K*. All elements stored in the right subtree of a node with value *K* have values >= *K*.



BSTNode (1)

```
class BSTNode<K,E> implements BinNode<E> {
  private K key;
  private E element;
  private BSTNode<K,E> left;
  private BSTNode<K,E> right;

  public BSTNode() {left = right = null; }
  public BSTNode(K k, E val)
  { left = right = null; key = k; element = val; }
  public BSTNode(K k, E val,
                 BSTNode<K,E> l, BSTNode<K,E> r)
  { left = l; right = r; key = k; element = val; }

  public K key() { return key; }
  public K setKey(K k) { return key = k; }

  public E element() { return element; }
  public E setElement(E v) { return element = v; }
```

BSTNode (2)

```
  public BSTNode<K,E> left() { return left; }
  public BSTNode<K,E> setLeft(BSTNode<K,E> p)
    { return left = p; }

  public BSTNode<K,E> right() { return right; }
  public BSTNode<K,E> setRight(BSTNode<K,E> p)
    { return right = p; }

  public boolean isLeaf()
  { return (left == null) && (right == null); }
}
```

BST (1)

```
/** BST implementation for Dictionary ADT */
class BST<K extends Comparable<? super K>, E>
          implements Dictionary<K, E> {
  private BSTNode<K,E> root; // Root of BST
  private int nodecount;     // Size of BST

  /** Constructor */
  BST() { root = null; nodecount = 0; }

  /** Reinitialize tree */
  public void clear()
    { root = null; nodecount = 0; }

  /** Insert a record into the tree.
      @param k Key value of the record.
      @param e The record to insert. */
  public void insert(K k, E e) {
    root = inserthelp(root, k, e);
    nodecount++;
  }
```

BST (2)

```
/** Remove a record from the tree.
     @param k Key value of record to remove.
     @return Record removed, or null if
      there is none. */
 public E remove(K k) {
   E temp = findhelp(root, k);   // find it
   if (temp != null) {
     root = removehelp(root, k); // remove it
     nodecount--;
   }
   return temp;
 }
```

BST (3)

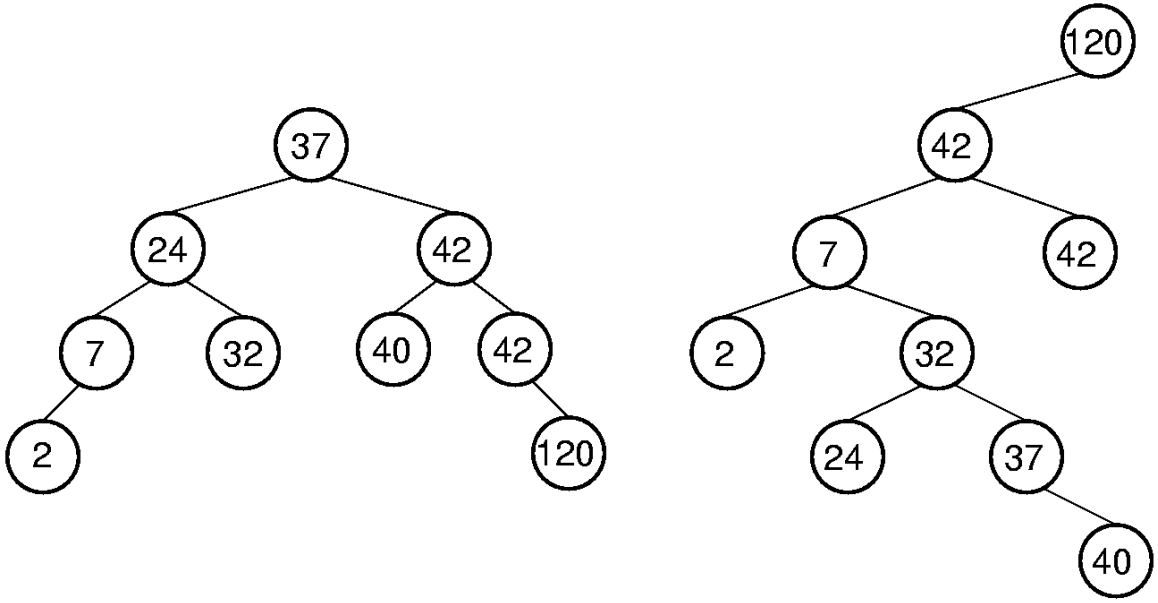
```
  /** Remove/return root node from tree.
    @return The record removed, null if empty. */
  public E removeAny() {
    if (root != null) {
      E temp = root.element();
      root = removehelp(root, root.key());
      nodecount--;
      return temp;
    }
    else return null;
  }

  /** @return Record with key k, null if none.
      @param k The key value to find. */
  public E find(K k)
    { return findhelp(root, k); }

  /** @return Number of records in dictionary. */
  public int size() { return nodecount; }
}
```

Recursion Example

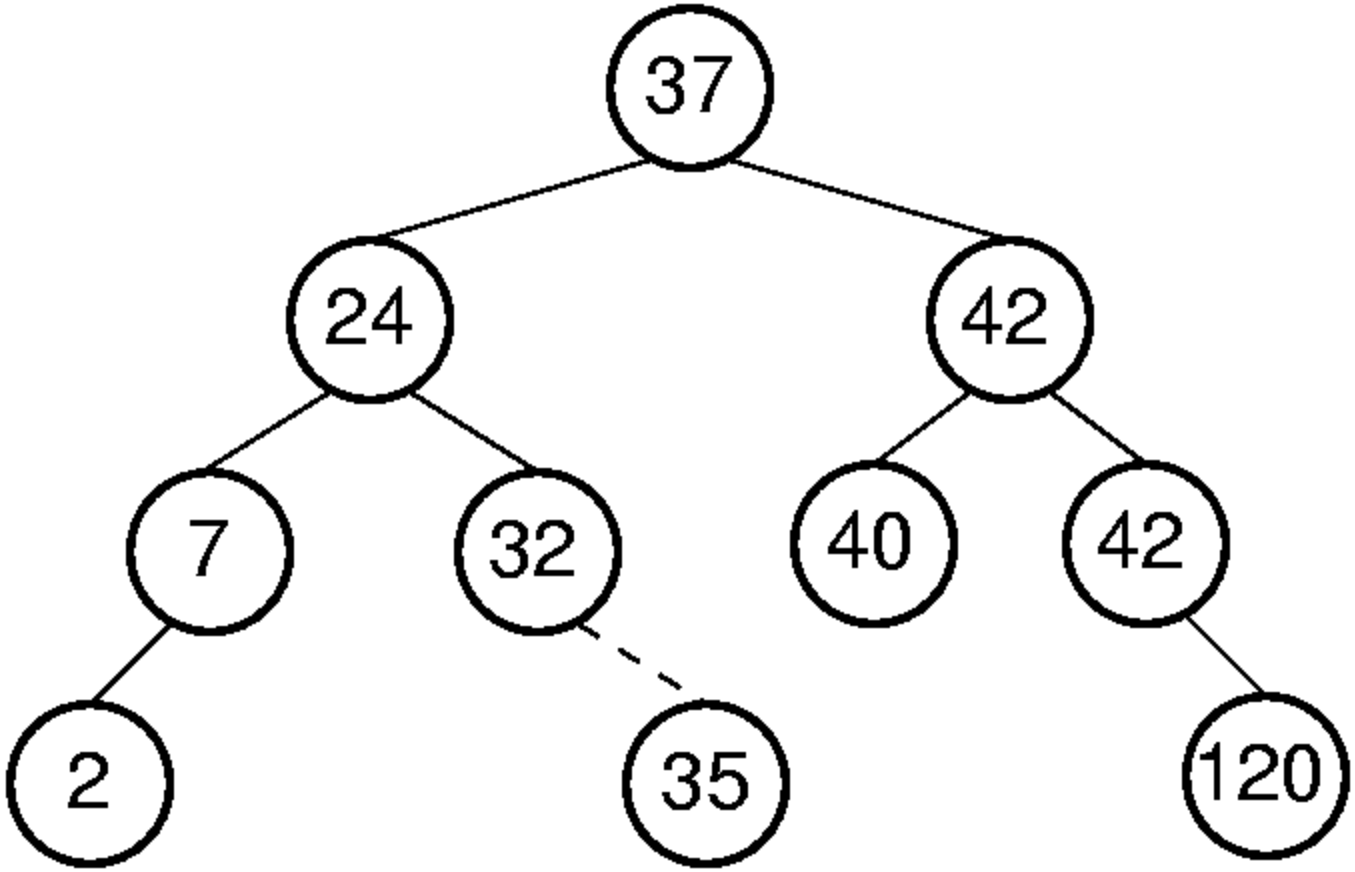
```
boolean checkBST(BSTNode<Integer> rt,
                 Integer low, Integer high) {
  if (rt == null) return true;
  Integer rootkey = rt.key();
  if ((rootkey < low) || (rootkey > high))
    return false; // Out of range
  if (!checkBST(rt.left(), low, rootkey-1))
    return false; // Left side failed
  return checkBST(rt.right(), rootkey, high);
}
```

BST Search

```
private E findhelp(BSTNode<K,E> rt, K k) {
  if (rt == null) return null;
  if (rt.key().compareTo(k) > 0)
    return findhelp(rt.left(), k);
  else if (rt.key().compareTo(k) == 0)
    return rt.element();
  else return findhelp(rt.right(), k);
}
```

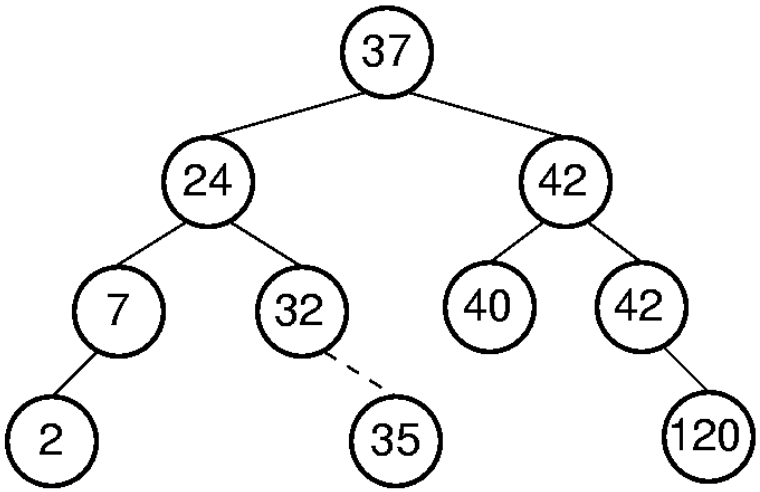


BST Insert (1)



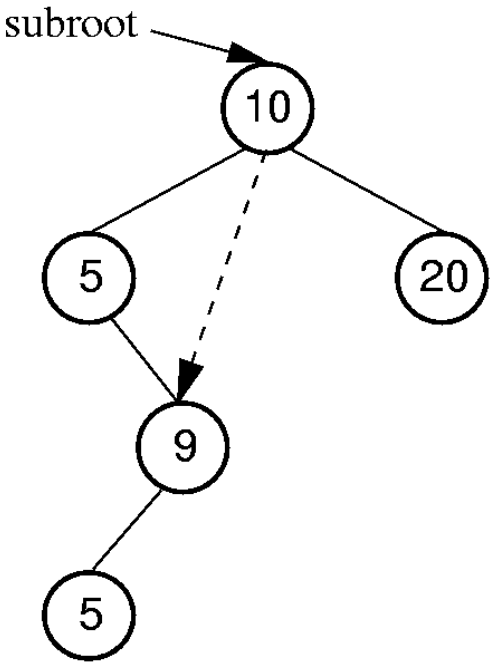
BST Insert (2)

```
private BSTNode<K,E>
      inserthelp(BSTNode<K,E> rt, K k, E e) {
  if (rt == null) return new BSTNode<K,E>(k, e);
  if (rt.key().compareTo(k) > 0)
    rt.setLeft(inserthelp(rt.left(), k, e));
  else
    rt.setRight(inserthelp(rt.right(), k, e));
  return rt;
}
```

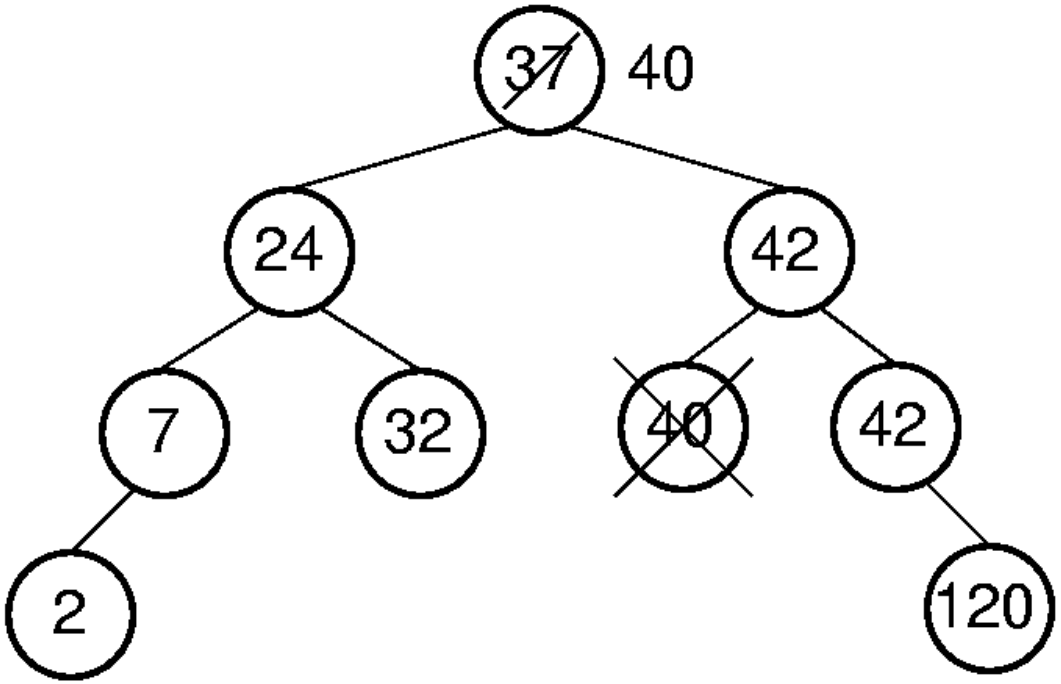


Get/Remove Minimum Value

```
private BSTNode<K,E>
        getmin(BSTNode<K,E> rt) {
  if (rt.left() == null)
    return rt;
  else return getmin(rt.left());
}

private BSTNode<K,E>
    deletemin(BSTNode<K,E> rt) {
  if (rt.left() == null)
    return rt.right();
  else {
    rt.setLeft(deletemin(rt.left()));
    return rt;
  }
}
```



BST Remove (1)



- Main Idea (when the item to remove has 2 children)
  - Find the minimum element from the right subtree of the item to remove
  - Swap the minimum element with the item to remove

BST Remove (2)

```
/** Remove a node with key value k
    @return The tree with the node removed */
private BSTNode<K,E>
               removehelp(BSTNode<K,E> rt, K k) {
  if (rt == null) return null;
  if (rt.key().compareTo(k) > 0)
    rt.setLeft(removehelp(rt.left(), k));
  else if (rt.key().compareTo(k) < 0)
    rt.setRight(removehelp(rt.right(), k));
```

BST Remove (3)

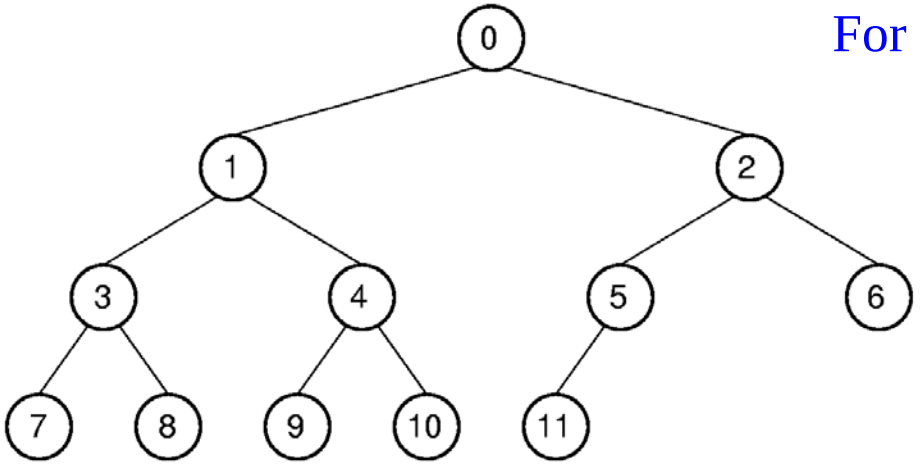
```
  else { // Found it, remove it
     if (rt.left() == null)
       return rt.right();
     else if (rt.right() == null)
       return rt.left();
     else { // Two children
       BSTNode<K,E> temp = getmin(rt.right());
       rt.setElement(temp.element());
       rt.setKey(temp.key());
       rt.setRight(deletemin(rt.right()));
     }
   }
   return rt;
}
```

Time Complexity of BST Operations

- Find: O(*d*)
- Insert: O(*d*)
- Delete: O(*d*)
- *d* = depth of the tree
- *d* is O(log *n*) if tree is balanced. What is the worst case?

Array Implementation (1)

For complete binary tree



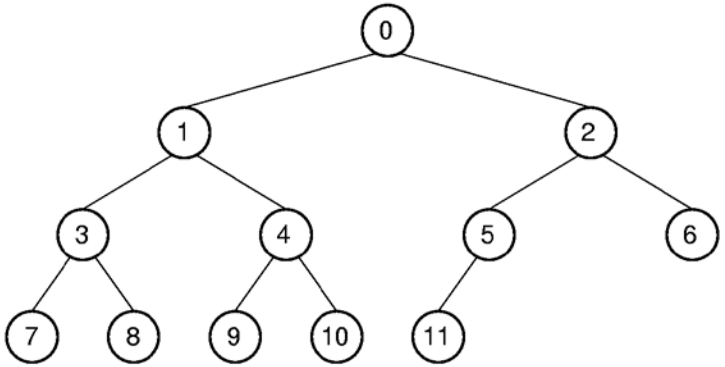
Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

Array Implementation (2)

- Parent (r) =
- Leftchild(r) =
- Rightchild(r) =
- Leftsibling(r) =
- Rightsibling(r) =

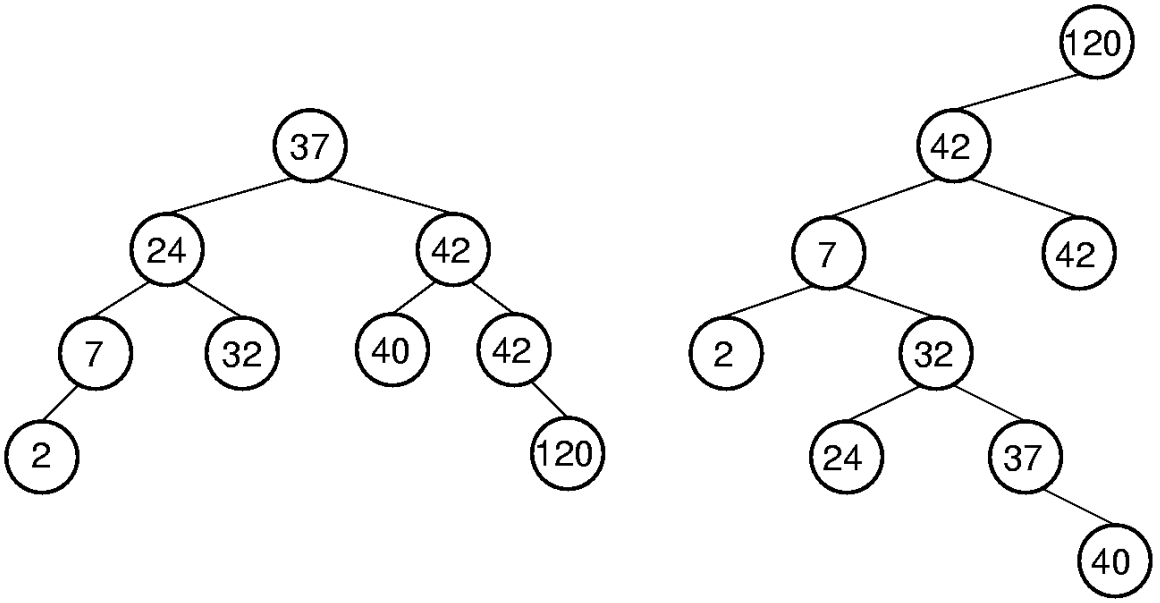
Array Implementation (3)

- Parent (*r*) = floor ((r-1)/2) if r != 0
- Leftchild(*r*) = 2r+1 if 2r+1 < n
- Rightchild(*r*) = 2r+2 if 2r+2 < n
- Leftsibling(*r*) = r – 1 if r is even
- Rightsibling(*r*) = r + 1 if r is odd and r+1 < n



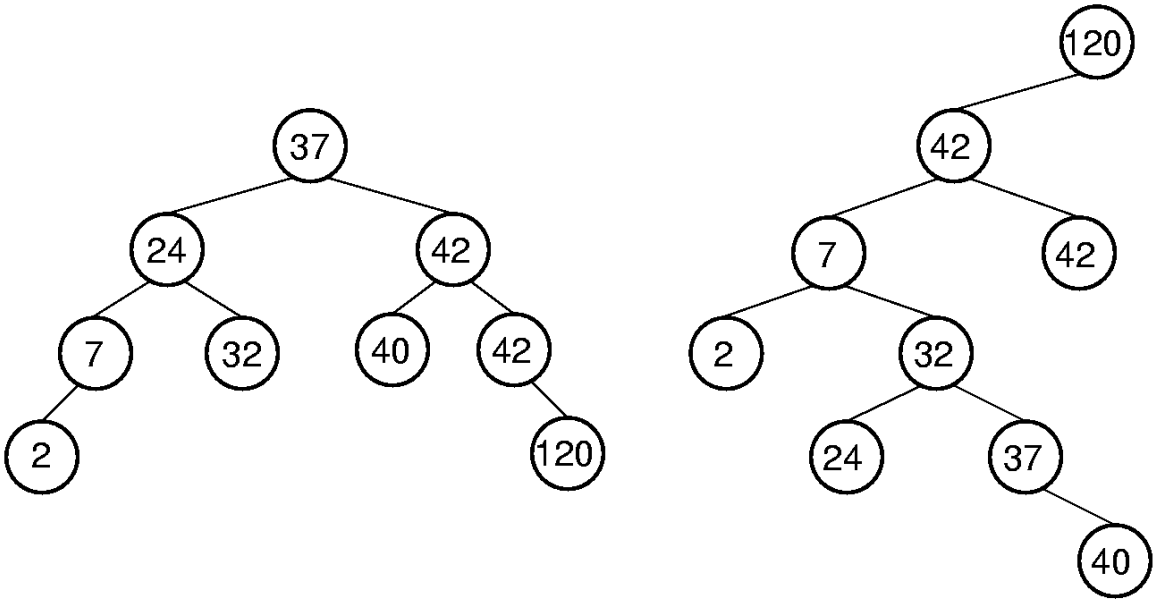
BST and Traversal

- What do we get from an inorder traversal from BST?



BST and Traversal

- What do we get from an inorder traversal from BST?
  - Sorted values (in increasing order) !



What you need to know

- Implementations and space overhead of binary tree
- How to implement link-based BST operations
  - Insert, remove, deletemin, find …
  - Get familiar with recursions in the operations
  - Time complexity of BST operations
- Array-based implementations for complete binary tree

Questions?