Binary Trees

Binary Tree ADT

- A recursive data structure with a root node and left and right children being roots of binary trees themselves
- Each internal node has at least one child and an external node (leaf) has no children
- Operations:

  leftChild(v) – left child of internal node v

  rightChild(v) – right child of internal node v

  isLeaf(v) – true iff v is a leaf or external node

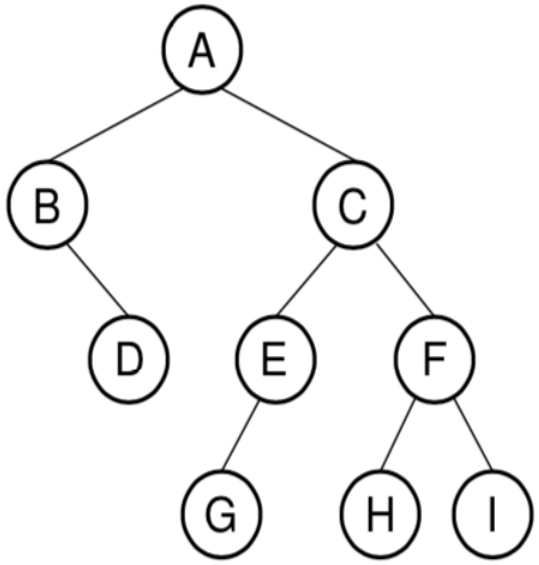


Figure 5.1 A binary tree. Node *A* is the root. Nodes *B* and *C* are *A*'s children. Nodes *B* and *D* together form a subtree. Node *B* has two children: Its left child is the empty tree and its right child is *D*. Nodes *A*, *C*, and *E* are ancestors of *G*. Nodes *D*, *E*, and *F* make up level 2 of the tree; node *A* is at level 0. The edges from *A* to *C* to *E* to *G* form a path of length 3. Nodes *D*, *G*, *H*, and *I* are leaves. Nodes *A*, *B*, *C*, *E*, and *F* are internal nodes. The depth of *I* is 3. The height of this tree is 4.

Source : Textbook

Notation

A *path* from n_1 to n_k: a sequence of nodes $n_1, n_2, \ldots, n_k$ in the tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$. The length of the path is $k - 1$.

If there is a path from node R to node M, then R is an *ancestor* of M, and M is a *descendant* of R.

The *depth* of a node M is the length of the path from the root of the tree to M.

The *height* of a tree is one more than the depth of the deepest node in the tree.

All nodes of depth d are at level d in the tree.

Source : Dr. Calvin & textbook

Full and Complete Binary Trees

- *Full* binary tree: Each node is either a leaf or internal node with exactly two non-empty children.
- *Complete* binary tree: If the height of the tree is h, then all leavels except possibly level $h - 1$ are completely full. The bottom level has all nodes to the left side.

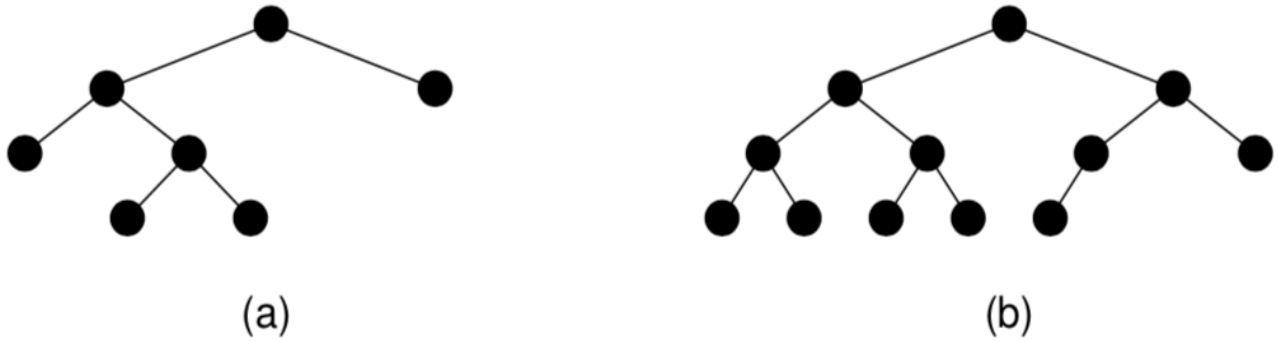


Figure 5.3 Examples of full and complete binary trees. (a) This tree is full (but not complete). (b) This tree is complete (but not full).

Source : Dr. Calvin & textbook

Uses of binary trees

- Extension called binary search trees are efficient data structures for ordered dictionaries or sets (keys can be sorted)
- Extension called heaps or priority queues are efficient data structures for a variety of applications
- Binary coding schemes (e.g. Huffman coding) for data compression use binary trees for 0-1 coding

Source : Dr. Calvin & textbook

Full Binary Tree Theorem

Theorem: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

Proof (by mathematical induction on number of internal nodes)

Base case: A full binary tree with 1 internal node must have two leaf nodes.

Induction Hypothesis: Assume any full binary tree T containing $n - 1$ internal nodes has n leaves.

Full Binary Tree Theorem Proof(contd.)

Induction Step: Given tree T with n internal nodes, pick internal node I with two leaf children. Remove I's children, call resulting tree T'.

By induction hypothesis, T' is a full binary tree with n leaves since it has $n - 1$ internal nodes.

Restore I's two children. The number of internal nodes has now gone up by 1 to reach n. The number of leaves has also gone up by 1 to $n + 1$.

Full Binary Tree Corollary

Theorem: The number of null pointers in a non-empty tree is one more than the number of nodes in the tree.

Proof: Replace all null pointers with a pointer to an empty leaf node. This is a full binary tree.

Source : Dr. Calvin & textbook

Some other facts about binary trees:

Theorem A: A binary tree with k levels has at most 2^{k-1} leaves. (Prove by strong induction on k.)

Theorem B: There are no more than 2^k nodes on level k of a binary tree.

Theorem C: A binary tree with k levels has at most $2^{k+1} - 1$ nodes.

Theorem D: If a binary tree has n nodes, then the number of levels is at least $\lceil \lg(n+1) \rceil - 1$. (Follows from C.)

Theorem E: A binary tree with L leaves has at least $\lceil \lg(L) + 1 \rceil$ levels. (Follows from A.)

Source : Dr. Calvin & textbook

```
/** ADT for binary tree nodes */
public interface BinNode<E> {
 /** Return and set the element value */
 public E element();
 public E setElement(E v);

 /** Return the left child */
 public BinNode<E> left();
 /** Return the right child */
 public BinNode<E> right();

 /** Return true if this is a leaf node */
 public boolean isLeaf();
}
```

Source : Dr. Calvin & textbook

Binary tree traversals

Any process for visiting the nodes in some order is called a traversal.

Any traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes.

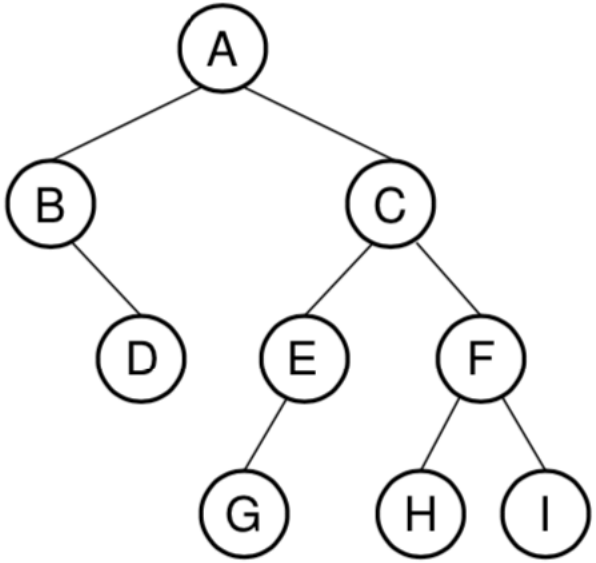
- Preorder traversal: Visit each node before visiting its children.
- Postorder traversal: Visit each node after visiting its children.
- Inorder traversal: Visit the left subtree, then the node, then the right subtree.

Source : Dr. Calvin & textbook

BT traversals

```
/** @param r The root of the subtree */

void preorder(BinNode r)
{
 if (r == null) return; // Empty subtree
 visit(r);
 preorder(r.left());
 preorder(r.right());
}
void inorder(BinNode r)
{
 if (r == null) return; // Empty subtree
 inorder(r.left());
 visit(r);
 inorder(r.right());
}
void postorder(BinNode r)
{
 if (r == null) return; // Empty subtree
 postorder(r.left());
 postorder(r.right());
 visit(r);
}
```

BT traversal example



Preorder enumeration : ABDCEGFHI

Inorder enumeration : BDAGECHFI

Postorder enumeration : DBGEHIFCA

Source : Dr. Calvin & textbook

Recursive function example for BT

```
public class BinaryTree<E> {
    private BinNode<E> root;
    private int count(BinNode<E> localRoot) {
        return localRoot == null ? 0 :
                1 + count(localRoot.left()) + count(localRoot.right());
    }

    public int count() {
        return count(root);
    }
}
```

Source : Dr. Calvin & textbook

Binary search trees

- Binary trees having ordering property:

  For any node v, all keys in left subtree of node (if it exists) have items smaller than the item at v and all keys in right subtree of node (if it exists) have items larger than the item at v

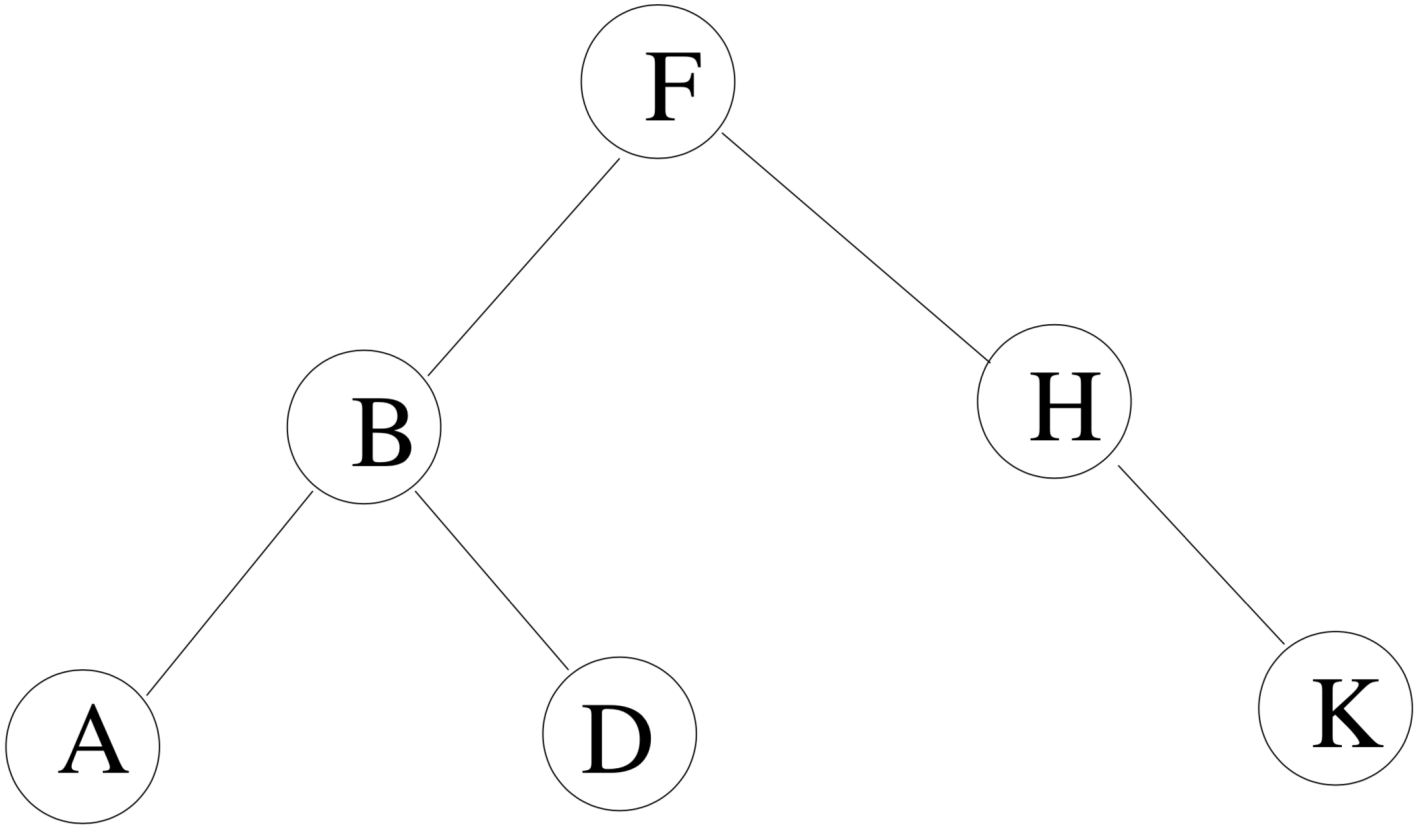
- Implements ordered dictionary or set where keys can be ordered
- Java :

  TreeMap<K,V> where K is key type and V is record or value type

  Implements Map<K,V> interface

Source : Dr. Calvin & textbook

Example BST



Source : Dr. Calvin & textbook

```
/** Binary tree node implementation: Pointers to children
    @param E The data element
    @param Key The associated key for the record */
class BSTNode<Key, E> implements BinNode<E> {
  private Key key;               // Key for this node
  private E element;             // Element for this node
  private BSTNode<Key,E> left;  // Pointer to left child
  private BSTNode<Key,E> right; // Pointer to right child

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

  /** Get and set the key value */
  public Key key() { return key; }
  public void setKey(Key k) { key = k; }

  /** Get and set the element value */
  public E element() { return element; }
  public void setElement(E v) { element = v; }

  /** Get and set the left child */
  public BSTNode<Key,E> left() { return left; }
  public void setLeft(BSTNode<Key,E> p) { left = p; }

  /** Get and set the right child */
  public BSTNode<Key,E> right() { return right; }
  public void setRight(BSTNode<Key,E> p) { right = p; }

  /** @return True if a leaf node, false otherwise */
  public boolean isLeaf()
  { return (left == null) && (right == null); }
}
```

Figure 5.7 A binary tree node class implementation.

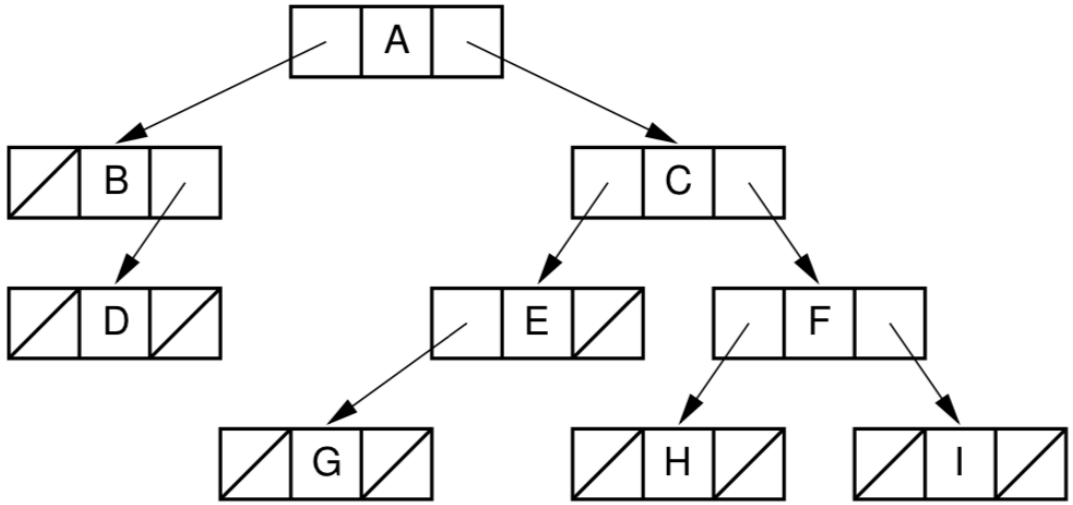


Figure 5.8 Illustration of a typical pointer-based binary tree implementation, where each node stores two child pointers and a value.

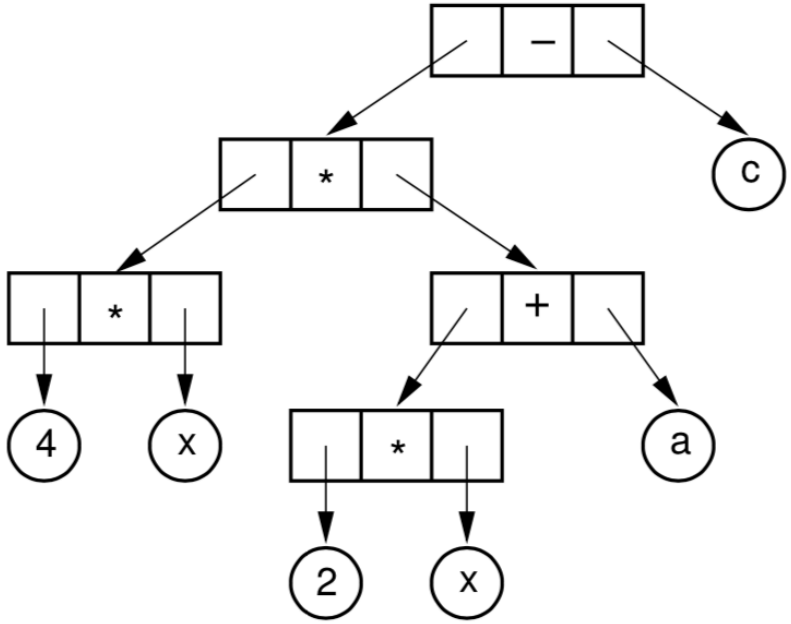


Figure 5.9 An expression tree for $4x(2x + a) - c$.

Dictionary operations for BST – find()

```
public class BST<Key,E> {
    private BSTNode<Key,E> root;
    private E findAux(BSTNode<Key,E> localRoot, Key searchKey) {
        if (localRoot == null) { return null; }
        int result = searchKey.compareTo(localRoot.key);
        if (result == 0) { return localRoot. element; }
        return result < 0 ? findAux(localRoot.left, searchKey)
                          : findAux(localRoot.right, searchKey);
    }
    public E find(Key searchKey) {
         return findAux(root, searchKey);
    }
}
```

Source : Dr. Calvin & textbook

Dictionary operations for BST - insert

```
public class BST<Key,E> {

    private int size;

private BSTNode<Key,E> insertAux(BSTNode<Key,E> localRoot, Key
key, E value) {

        if (localRoot == null) { size++; return BSTNode<>(key,value); }

        int result = key.compareTo(localRoot.key);

        if (result < 0) {

           localRoot.setLeft(insertAux(localRoot.left, key,value);

        else {

           localRoot.setRight(insertAux(localRoot.right, key,value);

         }

         return r;

}
```

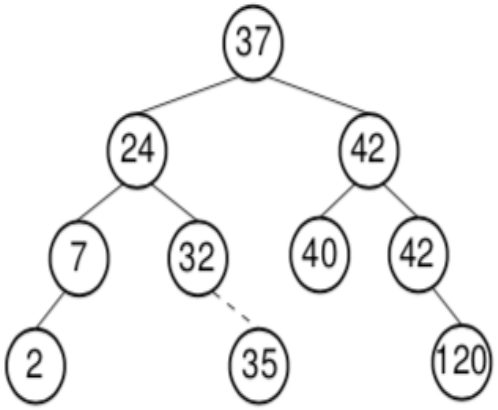
Source : Dr. Calvin & textbook

Dictionary operations for BST

```
public class BST<Key,E> {

…….

….

..

public void insert(Key key,  E value) {
        root = insertAux(root, key, value);
    }
}
```

Source : Dr. Calvin & textbook

BST key insertion example



Source : Dr. Calvin & textbook