GIUSEPPE TURINI

CS-102: COMPUTING AND ALGORITHMS 2
LESSON 06
TREES

Kettering
UNIVERSITY

HIGHLIGHTS

Trees Terminology and Properties

ADT Binary Tree (BT)

- Definition, Full BT, Complete BT, Balanced BT, and Min-Max Height
- Main Operations, Preorder/Inorder/Postorder Traversals
- Array-Based Implementation of a Binary Tree
- Reference-Based Implementation of a Binary Tree
- Tree Traversals using Iterators

Binary Search Trees (BSTs)

- Definition, Records and Fields and Key, Key Implementation
- Main Operations, Search-Retrieval/Insertion/Deletion/Traversal
- Reference-Based Implementation, Efficiency, Treesort Sorting Algorithm
- Saving/Loading a BST into/from File, BSTs in the JCF

N-ary Trees

STUDY GUIDE

STUDY MATERIAL

- **This slides.**

SELECTED EXERCISES

- **Set 1:** ex. 3-12, ex. 15, ex. 19, ex. 23, ex. 26-27, ex. 34.
- **Set 2:** ex. 1-2, ex. 4a-g, ex. 6a-e, ex. 8-12, ex. 13a-h, ex. 14a-h, ex. 30-32, ex. 37, ex. 40.

ADDITIONAL RESOURCES

- **"Object-Oriented Data Structures Using Java (4th Ed.)", chap. 7.**
- "Data Abstraction and Problem Solving with Java (3rd Ed.)", chap. 11.
- visualgo.net/en/bst

TREES - TERMINOLOGY AND PROPERTIES 1

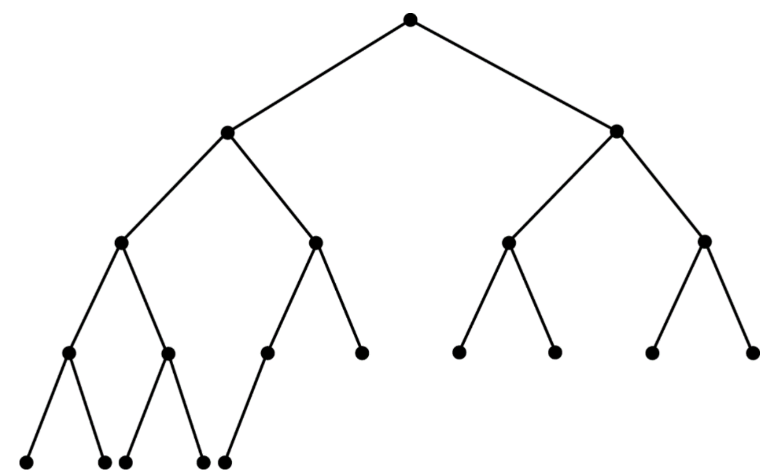
GENERAL TREE

A general tree **T** is a set of nodes with a **hierarchical structure** ("parent-child" relationships between nodes) such that **T** is partitioned into disjoint subsets:

- a single node **r** (called the **root**), and
- subsets that are general trees (called **subtrees** of **r**).

SUBTREE

A subtree **S** of node **n**, is a tree that consists of a child **c** (if any) of node **n** and all the descendant nodes of the child node **c**.



TREES - TERMINOLOGY AND PROPERTIES 2

Parent Node: The parent node **p** of node **n**, is the node directly above **n** in the tree **T**.

Child Node: A child node **c** of node **n**, is a node directly below node **n** in the tree **T**.

Root Node: The root node **r** of a tree **T**, is the only node in **T** with no parent node.

Leaf Node: A leaf node **l** of a tree **T**, is a node with no child nodes.

Sibling Nodes: Sibling nodes, are nodes with a common parent node.

Ancestor Node: An ancestor node **a** of node **n**, is a node on path from root **r** to **n**.

Descendant Node: A descendant node **d** of node **n**, is a node on the path from node **n** to a leaf **l**.

TREES - TERMINOLOGY AND PROPERTIES 3

LEVEL OF A NODE IN A TREE

The level **i** of a node **n** in a tree **T** can be defined as follows:

- **if n == r** (i.e. **n** root of **T**), then **n** is at level **1** (i.e. **i = 1**);
- **if n != r** (i.e. **n** not root of **T**), then **n** is at level **i = 1 + (level of parent node)**.

HEIGHT (OR DEPTH) OF A TREE

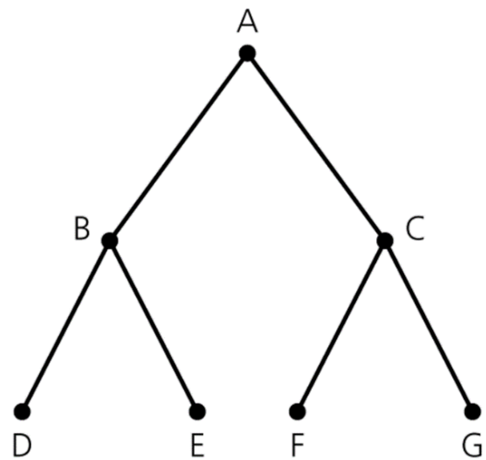
The height **h** of a tree (aka as its depth **d**), is the number of nodes on the longest path from the root node **r** to a leaf node **l**, or alternatively:

- **if T is empty**, its height **h** is **0** (i.e. **h = 0**);
- **if T is not empty**, its height **h** = **maximum level of its nodes**.

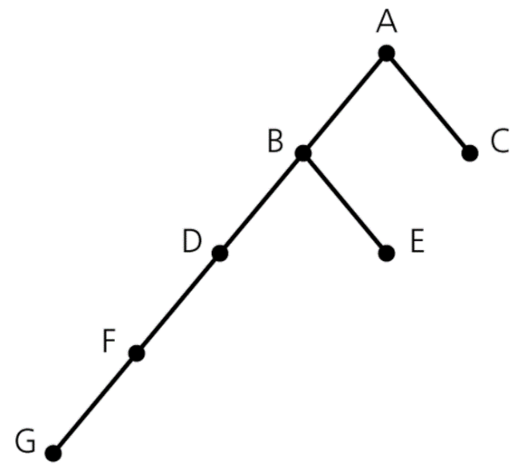
TREES - TERMINOLOGY AND PROPERTIES 4

HEIGHT OF A TREE: EXAMPLE

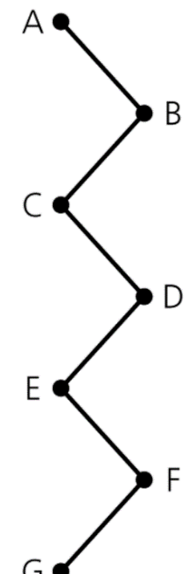
Trees with the same nodes but different heights (or depths): **(a)** a tree with height h = 3, **(b)** a tree with height h = 5, and **(c)** a tree with height h = 7.



(a)



(b)



(c)

BINARY TREE - DEFINITION

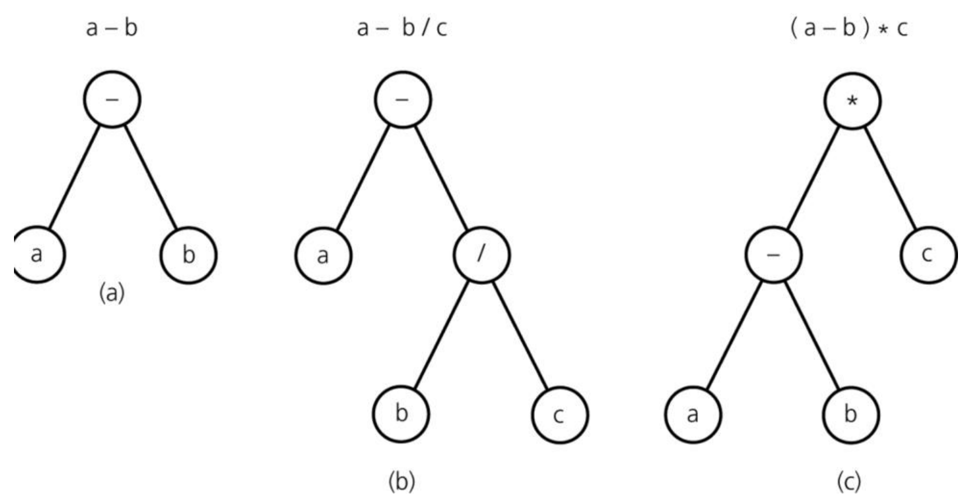
BINARY TREE

A binary tree is a set **T** of nodes such that either:

- **T is empty**, or
- **T is partitioned into 3 disjoint subsets:**
    - a single node **r**, called the **root**;
    - 2 subsets (binary trees), called **left** ($\mathbf{T_L}$) and **right** ($\mathbf{T_R}$) **subtrees** of **r**.

BINARY TREE: EXAMPLE

(a), **(b)**, and **(c)** are 3 binary trees representing algebraic expressions in infix form.



BINARY TREE - FULL BINARY TREE

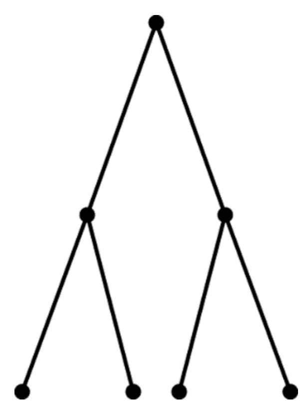
FULL BINARY TREE

The following is a **recursive definition of a full binary tree**:

- **if T is empty**, then **T** is a full binary tree of height 0;
- **if T is not empty and has height h > 0,** then **T** is a full binary tree if:
  - its **root subtrees** are both full binary trees of height **h – 1**.

FULL BINARY TREE: EXAMPLE

A full binary tree of height **3**.



BINARY TREE - COMPLETE BINARY TREE

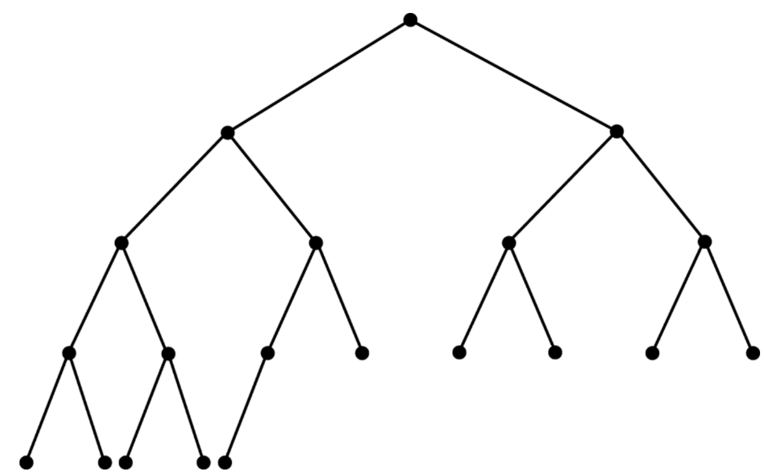
COMPLETE BINARY TREE

A binary tree **T** of height **h** is **complete** if:

- all nodes at level **h – 2** and above have **2** children each, and
- if a node at level **h – 1** has children,
    - then **all nodes to its left at the same level** have **2** children each, and
- if a node at level **h – 1** has **1** child, then it is a **left child**.

COMPLETE BINARY TREE: EXAMPLE

A complete binary tree.



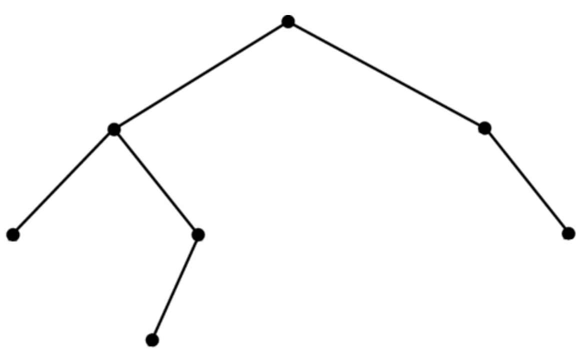
BINARY TREE - BALANCED BT

BALANCED BINARY TREE

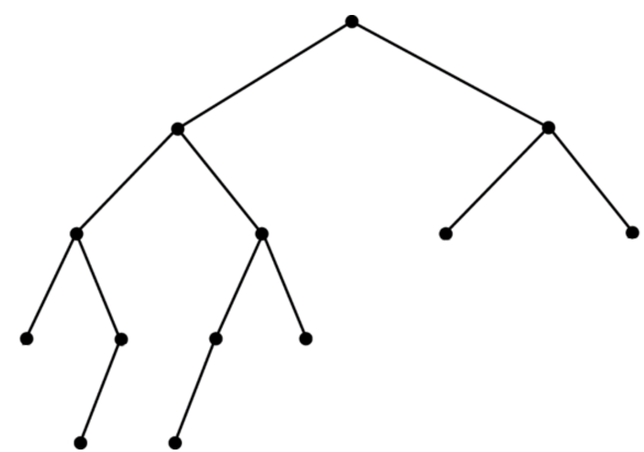
A binary tree is **balanced** if left and right subtrees of every node differ in height by no more than **1**.

Full binary trees are complete, and complete binary trees are balanced.

A Balanced Tree



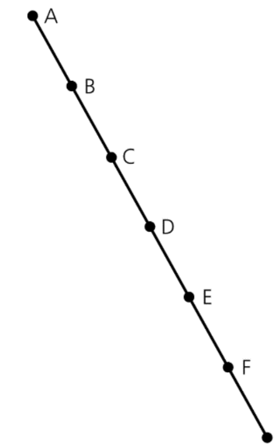
An Unbalanced Tree



BINARY TREE - MIN-MAX HEIGHT 1

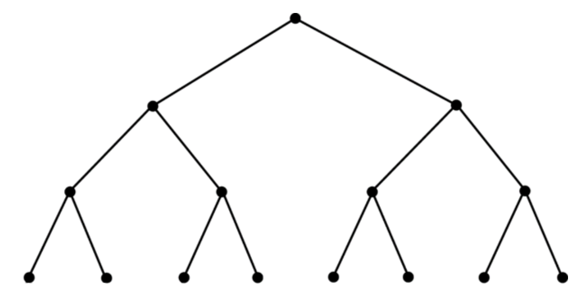
MAXIMUM HEIGHT OF A BINARY TREE

A binary tree with **n** nodes has **maximum height = n**
(when the tree structure is a continuous chain of nodes).



MINIMUM HEIGHT OF A BINARY TREE

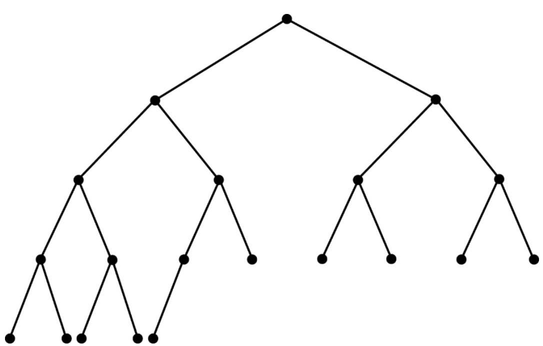
A binary tree with **n** nodes has
minimum height = ceiling($\log_2$(n+1))
(when the tree is perfectly balanced).

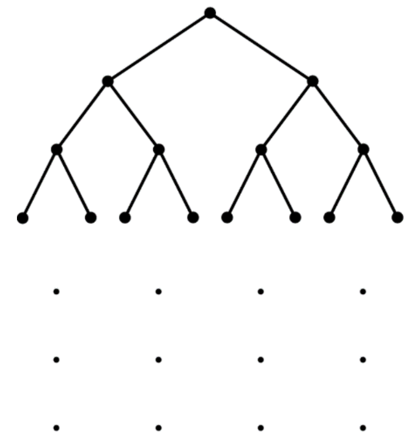


BINARY TREE - MIN-MAX HEIGHT 2

MINIMUM HEIGHT OF A BINARY TREE

Proof: To build a binary tree of **n** nodes with minimum height, we have to pack as many nodes as possible in upper levels, before moving on to the next level.
So, the tree takes the form below.





Level	Number of nodes at this level	Number of nodes at this and previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
.	.	.
.	.	.
.	.	.
h	2^{h-1}	$2^h - 1$

BINARY TREE - MIN-MAX HEIGHT 3

MINIMUM HEIGHT OF A BINARY TREE

Proof (continued): For a binary tree of height **h**, we can find the maximum number of nodes **n** (occuring when the tree is a **full binary tree**):

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = \sum_{k=0}^{h-1} 2^k = 2^h - 1$$

then solve for the height **h**: $n + 1 = 2^h \quad \Rightarrow \quad \log_2(n+1) = \log_2\left(2^h\right) = h$

This formula is not valid for all **n**, since **$\log_2(n)$** gives **non-integer values** for most **integer n** (i.e. for all but full binary trees), but height **h** has to be an integer value, so:

$$\lceil \log_2(n+1) \rceil = \text{minimum height of a binary tree}$$

BINARY TREE - MAIN OPERATIONS 1

BASIC OPERATIONS OF THE ADT BINARY TREE

The operations available for a particular ADT binary tree depend on the type of binary tree being implemented.

The following are the basic **operations that are common to all implementations** of the ADT binary tree:

```
// Pseudocode for the basic operations of the ADT binary tree.
void createBinaryTree(); // Creates an empty binary tree.
void createBinaryTree( TreeItemType rootItem ); // Creates a one-node binary tree.
void makeEmpty(); // Removes all of the nodes from a binary tree, leaving an empty tree.
boolean isEmpty(); // Determines whether a binary tree is empty.
TreeItemType getRootItem() throws TreeException; // Retrieves data in binary tree root.
void setRootItem( TreeItemType rootItem ) throws UnsupportedOperationException; // Set...
```

BINARY TREE - MAIN OPERATIONS 2

GENERAL OPERATIONS OF THE ADT BINARY TREE

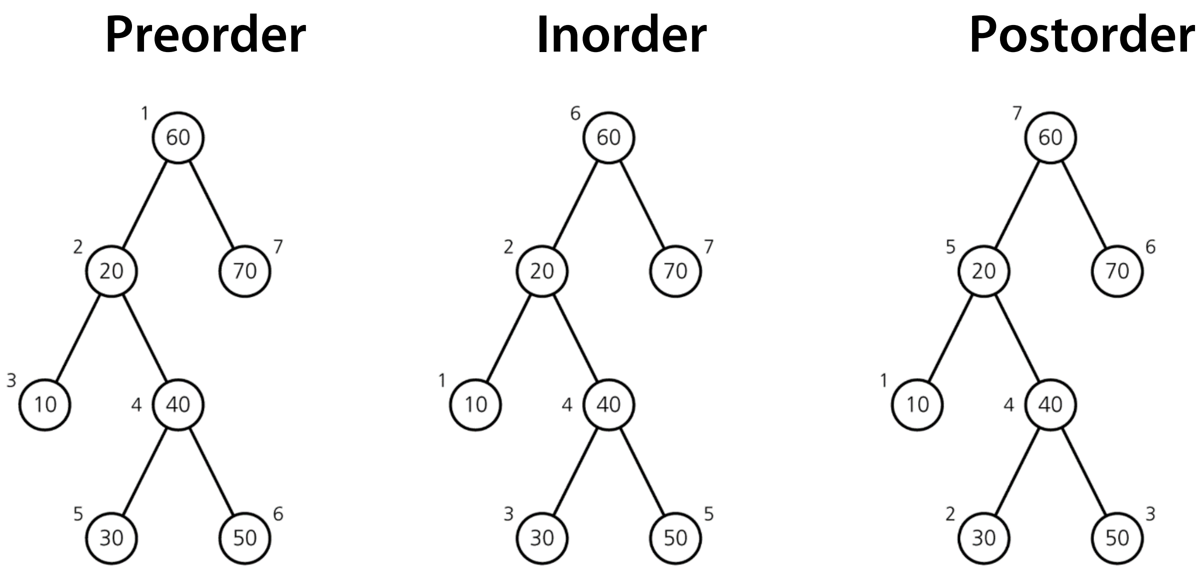
The following are the general operations of the ADT binary tree (added to the basic operations previously listed):

```
// Pseudocode for the general operations of the ADT binary tree.
void createBinaryTree(TreeItemType rootItem, BinaryTree leftTree, BinaryTree rightTree);
void setRootItem( TreeItemType newItem ); // Replaces data item in binary tree root.
void attachLeft( TreeItemType newItem ) throws TreeException; // Add left child to root.
void attachRight( TreeItemType newItem ) throws TreeException; // Add right child...
void attachLeftSubtree( BinaryTree leftTree ) throws TreeException; // Add left subtree.
void attachRightSubtree( BinaryTree rightTree ) throws TreeException; // Add...
BinaryTree detachLeftSubtree() throws TreeException; // Remove and returns left subtree.
BinaryTree detachRightSubtree() throws TreeException; // Remove and returns right...
```

BINARY TREE - TRAVERSALS 1

TRAVERSAL OF A BINARY TREE A

A traversal algorithm for a binary tree visits each node in the tree, and the followings are the main recursive traversal algorithms:



(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

BINARY TREE - TRAVERSALS 2

TRAVERSAL OF A BINARY TREE B

This is the code for a general **recursive traversal algorithm** for the ADT binary tree.

Note: We have different traversals depending on how the visit of the root node is arranged in respect to the subtrees traversals (**recursive calls**):

```
// Pseudocode for the traversal of the ADT binary tree.
void traverse( BinaryTree binTree ) {
   if( !binTree.isEmpty() ) {
      TreeItemType root = binTree.getRootItem();
      // Visit root node here (preorder traversal).
      traverse( root.getLeftSubtree() );
      // Visit root node here (inorder traversal).
      traverse( root.getRightSubtree() );
      // Visit root node here (postorder traversal).
   }
}
```

BINARY TREE - TRAVERSALS 3

TRAVERSAL OF A BINARY TREE C

Each of these traversals (preorder, inorder, and postorder) of a binary tree **visits every node exactly once**:

- thus, **n** node visits occur for a binary tree of **n** nodes;
- each node visit performs the same operations, independently of **n**, so it is **O(1)**;
- so, each binary tree traversal is in: $\mathbf{n \cdot O(1) = O(n)}$.

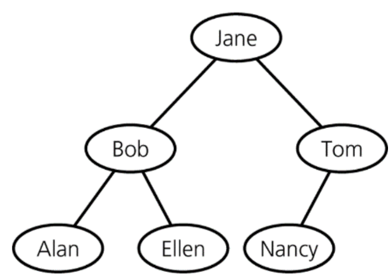
BINARY TREE - ARRAY-BASED BTs 1

ARRAY-BASED REPRESENTATION OF A BINARY TREE A

An array-based representation that works as follows:

- a class to define a node in the tree;
- a binary tree represented by an array of nodes;
- each node stores data and 2 indices for children;
- requires a **free list** to track available nodes.

(a)



(b)

tree

	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	–1
3	Alan	–1	–1
4	Ellen	–1	–1
5	Nancy	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
⋮	⋮	⋮	⋮

Free list (rows 6 onward)

root
0

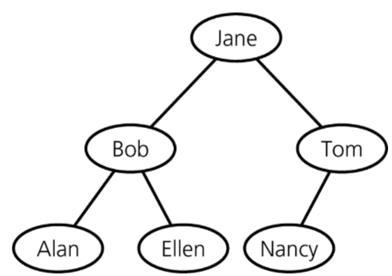
free
6

BINARY TREE - ARRAY-BASED BTs 2

ARRAY-BASED REPRESENTATION OF A BINARY TREE B

- **root** is an index to the tree root in the array (if tree is empty, **root** is **-1**);
- both **leftChild** and **rightChild** are indices (if node has no left/right child, that index is **-1**);
- **free** is index of first available node for insertion.

(a)



(b)

tree

	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	–1
3	Alan	–1	–1
4	Ellen	–1	–1
5	Nancy	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
⋮	⋮	⋮	⋮

Free list

root: 0

free: 6

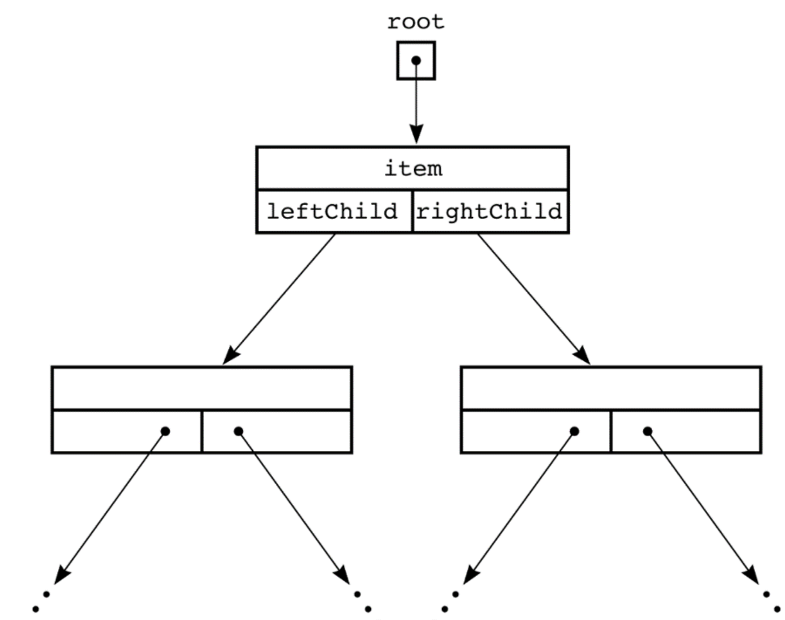
BINARY TREE - REFERENCE-BASED BTs 1

REFERENCE-BASED REPRESENTATION OF A BINARY TREE

In a reference-based representation of a binary tree, Java references can be used to link the nodes in the tree.

Classes for the reference-based implementation of the ADT binary tree:

- **TreeNode:** binary tree node.
- **TreeException:** exception class.
- **BinaryTreeBasis:** abstract class.
- **BinaryTree:** binary tree class.



BINARY TREE - REFERENCE-BASED BTs 2

TREE NODE

```java
package Tree;

class TreeNode<T> {
   T item; // Data item.
   TreeNode<T> leftChild; // Reference to the left child node.
   TreeNode<T> rightChild; // Reference to the right child node.

   // Constructors.

   public TreeNode( T newItem ) { item = newItem; leftChild = null; rightChild = null; }

   public TreeNode( T newItem, TreeNode<T> left, TreeNode<T> right ) {
      item = newItem;
      leftChild = left;
      rightChild = right;
   }
}
```

BINARY TREE - REFERENCE-BASED BTs 3

TREE EXCEPTION

```
package Tree;

import java.lang.RuntimeException;
import java.lang.String;

// Runtime exception for the ADT binary tree.
public class TreeException extends java.lang.RuntimeException {

   // Constructor.
   public TreeException( String s ) { super(s); }

}
```

BINARY TREE - REFERENCE-BASED BTs 4

BINARY TREE BASIS A

```
package Tree;

// Abstract class for binary trees, used for inheritance purposes only.
// Note: no direct instances of this class!
public abstract class BinaryTreeBasis<T> {

   protected TreeNode<T> root; // Protected so only subclasses have direct access.

   // Constructors.
   public BinaryTreeBasis() { root = null; }
   public BinaryTreeBasis( T rootItem ) { root = new TreeNode<T>( rootItem ); }

   // Checks if the binary tree is empty.
   public boolean isEmpty() { return root == null; }

   // Removes all the nodes from the binary tree.
   public void makeEmpty() { root = null; }
```

BINARY TREE - REFERENCE-BASED BTs 5

BINARY TREE BASIS B

```
    // Returns the item in the root of the binary tree.
    public T getRootItem() throws TreeException {
        if( root == null ) { throw new TreeException( "TreeException: empty tree!" ); }
        else { return root.item; }
    }

    // Throws UnsupportedOperationException if operation is not supported.
    public abstract void setRootItem( T newItem );

}
```

BINARY TREE - REFERENCE-BASED BTs 6

BINARY TREE A

```
package Tree;

// A reference-based implementation of the ADT binary tree.
public class BinaryTree<T> extends BinaryTreeBasis<T> {

   // Constructors.
   public BinaryTree() {}
   public BinaryTree( T rootItem ) { super( rootItem ); }
   public BinaryTree( T rootItem, BinaryTree<T> leftTree, BinaryTree<T> rightTree ) {
      root = new TreeNode<T>( rootItem );
      attachLeftSubtree( leftTree );
      attachRightSubtree( rightTree );
   }

   // Protected constructor available only to class methods and subclasses,
   // to avoid exposing node references to clients!
   protected BinaryTree( TreeNode<T> rootNode ) { root = rootNode; }
```

BINARY TREE - REFERENCE-BASED BTs 7

BINARY TREE B

```
public void setRootItem( T newItem ) {
   if( root != null ) { root.item = newItem; }
   else { root = new TreeNode<T>( newItem ); }
}

public void attachLeft( T newItem ) {
   if( !isEmpty() && ( root.leftChild == null ) ) {
      root.leftChild = new TreeNode<T>( newItem ); }
}

public void attachRight( T newItem ) {
   if( !isEmpty() && ( root.rightChild == null ) ) {
      root.rightChild = new TreeNode<T>( newItem ); }
}
```

BINARY TREE - REFERENCE-BASED BTs 8

BINARY TREE C

```
public void attachLeftSubtree( BinaryTree<T> leftTree ) throws TreeException {
   if( isEmpty() ) { throw new TreeException( "TreeException: empty tree!" ); }
   else if( root.leftChild != null ) {
      throw new TreeException( "TreeException: cannot overwrite left subtree!" ); }
   else {
      root.leftChild = leftTree.root;
      leftTree.makeEmpty(); } // Warning: don't leave multiple entry points to tree!
}

public void attachRightSubtree( BinaryTree<T> rightTree ) throws TreeException {
   if( isEmpty() ) { throw new TreeException( "TreeException: empty tree!" ); }
   else if( root.rightChild != null ) {
      throw new TreeException( "TreeException: cannot overwrite right subtree!" ); }
   else {
      root.rightChild = rightTree.root;
      rightTree.makeEmpty(); } // Warning: don't leave multiple entry points to tree!
}
```

BINARY TREE - REFERENCE-BASED BTs 9

BINARY TREE D

```
   public BinaryTree<T> detachLeftSubtree() throws TreeException {
      if( isEmpty() ) { throw new TreeException( "TreeException: empty tree!" ); }
      else { // Create a new binary tree that has root left child as root node.
         BinaryTree<T> leftTree = new BinaryTree<T>( root.leftChild );
         root.leftChild = null;
         return leftTree; }
   }

   public BinaryTree<T> detachRightSubtree() throws TreeException {
      if( isEmpty() ) { throw new TreeException( "TreeException: empty tree!" ); }
      else { // Create a new binary tree that has root right child as root node.
         BinaryTree<T> rightTree = new BinaryTree<T>( root.rightChild );
         root.rightChild = null;
         return rightTree; }
   }
}
```

BINARY TREE - TRAVERSAL USING ITERATOR 1

TREE TRAVERSAL USING AN ITERATOR

The **TreeIterator** class implements the Java **Iterator** interface, and provides methods to set the iterator to the type of traversal desired. This class uses a **queue** to maintain the current traversal of the nodes in the tree.

An **iterative (non-recursive) traversal method** can be implemented using an explicit **stack** to mimic actions of a recursive call to **inorder**.

See: docs.oracle.com/javase/8/docs/api/java/util/iterator

BINARY TREE - TRAVERSAL USING ITERATOR 2

TREE ITERATOR A

```
package Tree;

import java.util.LinkedList;

// Iterator class for the ADT binary tree.
public class TreeIterator<T> implements java.util.Iterator<T> {
   private BinaryTreeBasis<T> binTree; // The binary tree used.
   private TreeNode<T> currNode; // Current node in the traversal.
   private LinkedList< TreeNode<T> > queue; // Queue for traversal sequence (see JCF).

   // Constructor.
   public TreeIterator( BinaryTreeBasis<T> bt ) {
      binTree = bt;
      currNode = null;
      // Empty queue means no traversal type selected, or end of current traversal.
      queue = new LinkedList< TreeNode<T> >();
   }
```

BINARY TREE - TRAVERSAL USING ITERATOR 3

TREE ITERATOR B

```
// JCF Iterator interface required methods.

public boolean hasNext() { return !queue.isEmpty(); }

public T next() throws java.util.NoSuchElementException {
   currNode = queue.remove();
   return currNode.item;
}

public void remove() throws UnsupportedOperationException {
   throw new UnsupportedOperationException();
}
```

BINARY TREE - TRAVERSAL USING ITERATOR 4

TREE ITERATOR C

```
// Traversal methods (preorder).

public void setPreorder() {
   queue.clear();
   preorder( binTree.root );
}

private void preorder( TreeNode<T> treeNode ) {
   if( treeNode != null ) {
      queue.add( treeNode );
      preorder( treeNode.leftChild );
      preorder( treeNode.rightChild ); }
}
```

BINARY TREE - TRAVERSAL USING ITERATOR 5

TREE ITERATOR D

```
// Traversal methods (inorder).

public void setInorder() {
   queue.clear();
   inorder( binTree.root );
}

private void inorder( TreeNode<T> treeNode ){
   if( treeNode != null ){
      inorder( treeNode.leftChild );
      queue.add( treeNode );
      inorder( treeNode.rightChild);}
}
```

BINARY TREE - TRAVERSAL USING ITERATOR 6

TREE ITERATOR E

```
   // Traversal methods (postorder).

   public void setPostorder() {
      queue.clear();
      postorder( binTree.root );
   }

   private void postorder( TreeNode<T> treeNode ){
      if( treeNode != null ){
         postorder( treeNode.leftChild );
         postorder( treeNode.rightChild);
         queue.add( treeNode ); }
   }

}
```

BINARY TREE - TRAVERSAL USING ITERATOR 7

TREE TEST A

```
import Tree.BinaryTree;
import Tree.TreeIterator;
import java.lang.String;

public class TreeTest {

   // ...
   public static void main( String[] args ) {
      // Build a binary tree (1).
      BinaryTree<String> bt1 = new BinaryTree<String>( "70" );
      // Build a binary tree (2).
      BinaryTree<String> bt2 = new BinaryTree<String>();
      bt2.setRootItem( "40" );
      bt2.attachLeft( "30" );
      bt2.attachRight( "50" );
```

BINARY TREE - TRAVERSAL USING ITERATOR 8

TREE TEST B

```
// Build a binary tree (3).
BinaryTree<String> bt3 = new BinaryTree<String>();
bt3.setRootItem( "20" );
bt3.attachLeft( "10" );
bt3.attachRightSubtree( bt2 );
// Build a binary tree (4).
BinaryTree<String> bt4 = new BinaryTree<String>( "60", bt3, bt1 );
// Setup a binary tree iterator.
TreeIterator<String> bt4Iter = new TreeIterator<String>( bt4 );
bt4Iter.setInorder(); // Init binary tree iterator.
// Use the binary tree iterator.
System.out.println( "---" );
while( bt4Iter.hasNext() ) { System.out.println( bt4Iter.next() ); }
System.out.println( "---" );
```

BINARY TREE - TRAVERSAL USING ITERATOR 9

TREE TEST C

```
        // Setup a subtree and the relative iterator.
        BinaryTree<String> bt4LeftTree = bt4.detachLeftSubtree();
        TreeIterator<String> bt4LeftTreeIter = new TreeIterator<String>( bt4LeftTree );
        // Iterate through the subtree.
        bt4LeftTreeIter.setInorder(); // Init binary tree iterator.
        System.out.println( "---" );
        while( bt4LeftTreeIter.hasNext() ) { System.out.println(bt4LeftTreeIter.next()); }
        System.out.println( "---" );
        // Iterate through the original binary tree (minus the detached subtree).
        bt4Iter.setInorder(); // Init binary tree iterator.
        // Use the binary tree iterator.
        System.out.println( "---" );
        while( bt4Iter.hasNext() ) { System.out.println( bt4Iter.next() ); }
        System.out.println( "---" );
    }

}
```

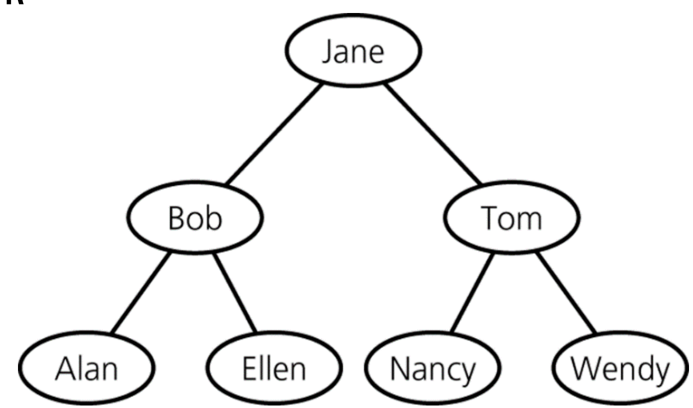
BST - DEFINITION

The search for a particular item in an ADT binary tree is not efficient.
This is corrected by the ADT binary search tree by organizing its data by value.

BINARY SEARCH TREE

A binary tree that has the following properties for each node **n**:

- value of node **n >** all values in its **left subtree T_L**;
- value of node **n <** all values in its **right subtree T_R**;
- both T_L and T_R are binary search trees.



Example: In figure, a binary search tree of names.

See: visualgo.net/en/bst

BST - RECORDS AND FIELDS AND KEY

A tree node may contain different data fields, so we have these definitions:

Records and Fields: A record is a group of fields (usually a class instance).

Search Key: A field (or a group of fields) able to **uniquely identify a record** is called search key, and it will need to be compared to the search key of other records.
Note: The search key should remain the same as long as the item is stored in the tree.

The abstract class **KeyedItem** extends **Comparable**, and contains the search key as a data field and a method for accessing the search key.
Note: KeyedItem must be extended by classes for items in a binary search tree!

See: docs.oracle.com/javase/8/docs/api/java/lang/comparable

BST - KEY IMPLEMENTATION

KEYED ITEM

```java
package Tree;

// Abstract class KeyedItem to store keys and enable key-key comparisons.
// Note: Use of lower bounded wildcards!
public abstract class KeyedItem< KT extends Comparable<? super KT > > {

   private KT searchKey; // The search key.

   // Constructor.
   public KeyedItem( KT key ) { searchKey = key; }

   // Accessor for the search key.
   public KT getKey() { return searchKey; }

   // Note: No modifier available for the search key, that can only be initialized!

}
```

BST - MAIN OPERATIONS

OPERATIONS OF THE ADT BINARY SEARCH TREE

- **Search-Retrieve** the item with a given search key from a binary search tree.
- **Insert** a new item into a binary search tree.
- **Delete** the item with a given search key from a binary search tree.
- **Traverse** the items in a binary search tree in preorder, inorder, or postorder.

See: visualgo.net/en/bst

Since the binary search tree is recursive in nature, it is natural to formulate recursive algorithms for its operations.

BST - OPERATION SEARCH-RETRIEVAL 1

OPERATIONS OF THE ADT BINARY SEARCH TREE: SEARCH

The following search algorithm searches the input binary search tree **bst** for an item with search key equal to the input search key **key**.

```
// Pseudocode of the search method for an ADT binary search tree.
public boolean search( BinarySearchTree bst, KeyType key ) {
   if( bst.isEmpty() ) { return false; } // If the tree is empty the key is not found.
   else if( key == bst.getRoot().getKey() ) { return true; } // Key found in tree root.
   // Compare input key to tree root key, and recursively search in proper subtree.
   else if( key < bst.getRoot().getKey() ) {
      return search( bst.getLeftSubtree(), key ); }
   else {
      return search( bst.getRightSubtree(), key ); }
}
```

Note: This **search** algorithm is the basis of **insertion**, **deletion**, and **retrieval**.
Note: The shape of the tree does not affect the validity of the search algorithm!

BST - OPERATION SEARCH-RETRIEVAL 2

OPERATIONS OF THE ADT BINARY SEARCH TREE: RETRIEVAL

The retrieval operation can be implemented by refining the **search** algorithm, that is: return the item with the desired search key if it exists, otherwise return null.

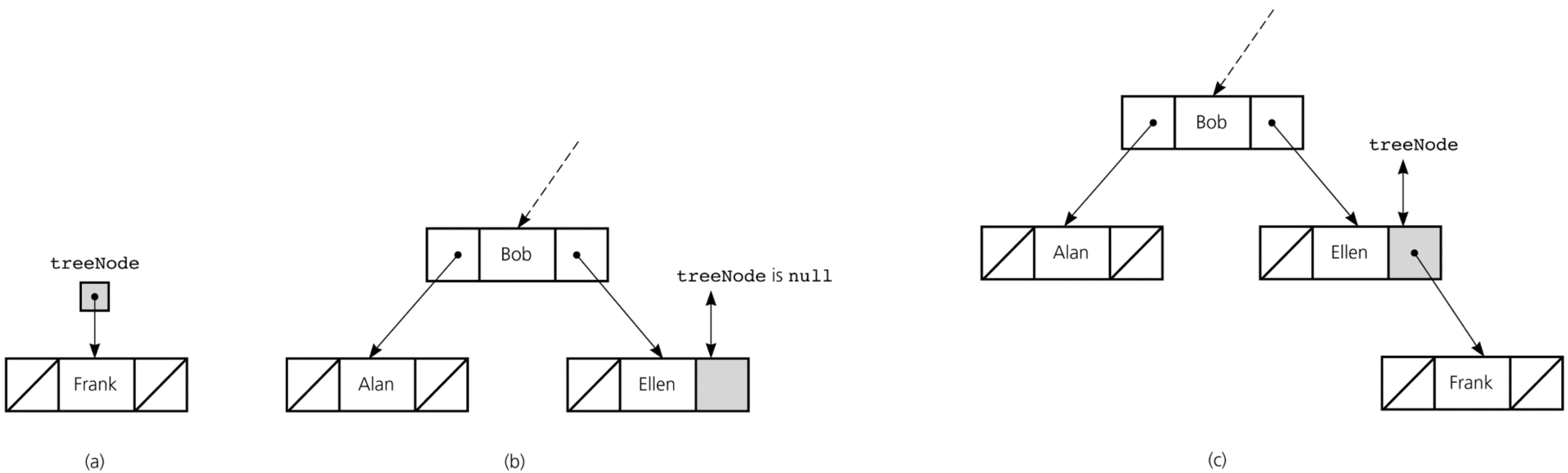
```
// Pseudocode of the retrieval method for an ADT binary search tree.
TreeItemType retrieveItem( TreeNode treeNode, KeyType searchKey ) {
   TreeItemType treeItem;
   if( treeNode == null ) { treeItem = null; }
   else if( searchKey == treeNode.item.getKey() ) { treeItem = treeNode.item; }
   else if( searchKey < treeNode.item.getKey() ) {
      treeItem = retrieveItem( treeNode.leftChild, searchKey ); }
   else { treeItem = retrieveItem( treeNode.rightChild, searchKey ); }
   return treeItem;
}
```

BST - OPERATION INSERTION 1

OPERATIONS OF THE ADT BINARY SEARCH TREE: INSERTION A

The insertion method for the binary search tree works as follows:

1. if the tree is empty: simply insert the new item **(a)**;
2. otherwise: insert the new item where the search method terminates **(b) (c)**.



BST - OPERATION INSERTION 2

OPERATIONS OF THE ADT BINARY SEARCH TREE: INSERTION B

```
// Pseudocode of the insertion method for an ADT binary search tree.
// Note: this method return a TreeNode to set the parent node children references!
public TreeNode insertItem( TreeNode node, TreeItemType item ) {
   if( node == null ) {
      node = new TreeNode( item, null, null ); }
   else if( item.getKey() < node.item.getKey() ) {
      node.leftChild = insertItem(node.leftChild, item); } // Recursion left subtree.
   else {
      node.rightChild = insertItem(node.rightChild, item); } // Recursion right subtree.
   return node; // Return the new node to allow parent to set children references.
}
```

Note: Take the output of a **preorder** traversal of a BST, and use it with this **insertItem** method to “clone” the original binary search tree replicating its content and shape!

BST - OPERATION DELETION 1

OPERATIONS OF THE ADT BINARY SEARCH TREE: DELETION A

The following are the steps required to delete a tree node:

1. use the search algorithm to locate the item with the specified key;
2. if the item is found, remove the item from the tree following step **(3)**;
3. three possible cases for node **n** containing the item to be deleted:
    a. if node **n** is a leaf:
        i. set the node **n** reference in its parent node **p** to **null**;
    b. if node **n** has only 1 child:
        i. let the parent node **p** of node **n** adopt the child node **c** of node **n**;
    c. if node **n** has 2 children:
        i. locate "another node **m** that is easier to remove than node **n**",
        ii. copy node **m** into node **n** (tree temporarily unsorted),
        iii. remove node **m** from the tree.

BST - OPERATION DELETION 2

OPERATIONS OF THE ADT BINARY SEARCH TREE: DELETION B

Question: What kind of node **m** is easier to remove than the node **n**?
Answer: A node that has no children or only 1 child.

Question: Can you choose any “easier” node **m** and copy its data into node **n**?
Answer: No, because you must preserve the sorting of the binary search tree.

Question: What data of node **m**, when copied into node **n**, will preserve the sorting?
Answer: You must choose a node **m** with a “search key **y** immediately after or immediately before search key **x**” of node **n** in the binary search tree sorted order.
If **y** is the key immediately after key **x**: then **y** is called **inorder successor** of **x**, and it is the search key of **the leftmost node in the right subtree of node n** (i.e. key **x**).

BST - OPERATION DELETION 3

OPERATIONS OF THE ADT BINARY SEARCH TREE: DELETION C

```
// Pseudocode of the deletion method for an ADT binary search tree (a).
public TreeNode deleteItem( TreeNode rootNode, KeyType searchKey ) {
   if( rootNode == null ) { throw new TreeException( "Item not found!" ); }
   else if( searchKey == rootNode.item.getKey() ) {
      TreeNode newRoot = deleteNode( rootNode, searchKey ); // Delete rootNode.
      return newRoot; } // Return new root node.
   else if( searchKey < rootNode.item.getKey() ) {
      TreeNode newLeft = deleteItem( rootNode.leftChild, searchKey );
      rootNode.leftChild = newLeft;
      return newRoot; } // Returns rootNode with new left subtree.
   else {
      TreeNode newRight = deleteItem( rootNode.rightChild, searchKey );
      rootNode.rightChild = newRight;
      return newRoot; } // Returns rootNode with new right subtree.
}
```

BST - OPERATION DELETION 4

OPERATIONS OF THE ADT BINARY SEARCH TREE: DELETION D

```
// Pseudocode of the deletion method for an ADT binary search tree (b).
private TreeNode deleteNode( TreeNode treeNode ) {
   if( treeNode.leftChild == null ) {
      if( treeNode.rightChild == null ) { return null; } // treeNode is a leaf.
      else { return treeNode.rightChild; } } // treeNode has only the right child.
   else if( treeNode.rightChild == null ) {
      return treeNode.leftChild; } // treeNode has only the left child.
   else { // treeNode has 2 children.
      // Find the inorder successor of treeNode key.
      TreeNode replacementItem = findLeftMost( treeNode.rightChild );
      TreeNode replacementRightChild = deleteLeftMost( treeNode.rightChild );
      treeNode.item = replacementItem.item;
      treeNode.rightChild = replacementRightChild;
      return treeNode; }
}
```

BST - OPERATION DELETION 5

OPERATIONS OF THE ADT BINARY SEARCH TREE: DELETION E

```
// Pseudocode of the deletion method for an ADT binary search tree (c).
private TreeNode findLeftMost( TreeNode treeNode ) {
   // Returns the node that is the leftmost descendant of the subtree rooted at treeNode.
   if( treeNode.leftChild == null ) { return treeNode; }
   else { return findLeftMost( treeNode.leftChild ); }
}

// Pseudocode of the deletion method for an ADT binary search tree (d).
private TreeNode deleteLeftMost( TreeNode treeNode ) {
   // Deletes leftmost descendant of treeNode. Returns subtree of deleted node.
   if( treeNode.leftChild == null ) { return treeNode.rightChild; }
   else {
      TreeNode replacementLeftChild = deleteLeftMost( treeNode.leftChild );
      treeNode.leftChild = replacementLeftChild;
      return treeNode; }
}
```

BST - OPERATION TRAVERSAL

OPERATIONS OF THE ADT BINARY SEARCH TREE: TRAVERSAL

Traversals for a binary search tree are the same as the traversals for a binary tree

```
// Pseudocode of the inorder traversal method for an ADT binary search tree.
public void inorder( BinarySearchTree bst ) {
   // Traverse the binary search tree in inorder.
   if( !bst.isEmpty() ) {
      inorder( bst.getRoot().getLeftSubtree() );
      System.out.println( bst.getRoot() ); // Do something with the root.
      inorder( bst.getRoot().getRightSubtree() ); }
}
```

Theorem: The **inorder** traversal of a binary search tree **T** will visit its nodes in **sorted search-key order**.

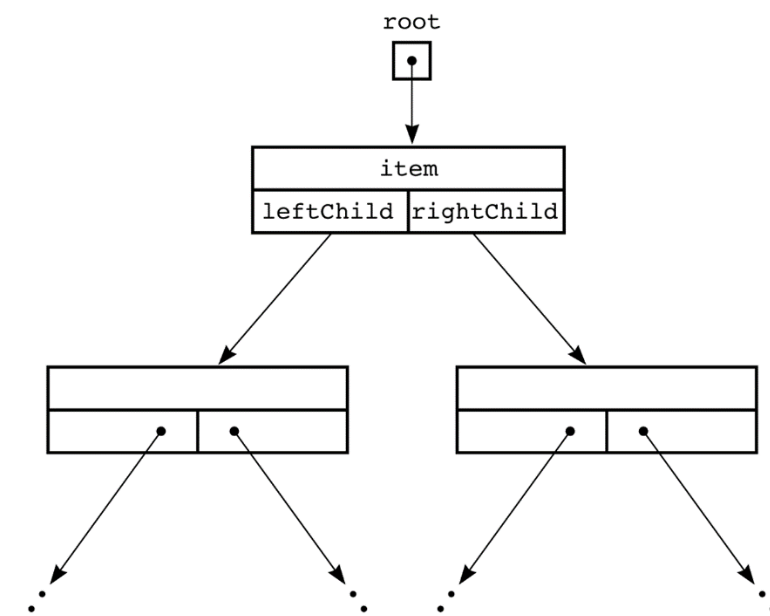
BST - REFERENCE-BASED IMPLEMENTATION 1

REFERENCE-BASED REPRESENTATION OF A BINARY SEARCH TREE A

In a reference-based representation of a binary search tree, Java references can be used to link the nodes in the tree.

Classes for a reference-based implementation of the ADT binary search tree:

- **TreeNode:** binary tree node.
- **TreeException:** exception class.
- **BinaryTreeBasis:** abstract class.
- **BinaryTree:** binary tree class.



BST - REFERENCE-BASED IMPLEMENTATION 2

REFERENCE-BASED REPRESENTATION OF A BINARY SEARCH TREE B

The **BinarySearchTree** class extends the **BinaryTreeBasis** class, inheriting the following methods:

- **isEmpty()**
- **makeEmpty()**
- **getRootItem()**
- and the constructors.

The **TreeIterator** class implements the Java **Iterator** interface, and provides methods to set the iterator to the type of traversal desired. This class uses a **queue** to maintain the current traversal of the nodes in the tree.
Note: The **TreeIterator** class can be used with the **BinarySearchTree** class.

See: docs.oracle.com/javase/8/docs/api/java/util/iterator

BST - REFERENCE-BASED IMPLEMENTATION 3

BINARY SEARCH TREE A

```
package Tree;

// Reference-based implementation of the ADT binary search tree.
// Assumption: A tree contains at most 1 item with a given search key at any time.
// Note: Use of lower bounded wildcards!
public class BinarySearchTree< T extends KeyedItem< KT >,
                                KT extends Comparable<? super KT > >
                                extends BinaryTreeBasis<T> {

   // Constructors.
   public BinarySearchTree() {}
   public BinarySearchTree( T rootItem ) { super( rootItem ); }
```

BST - REFERENCE-BASED IMPLEMENTATION 4

BINARY SEARCH TREE B

```
// Public methods.

public void setRootItem( T newItem ) throws UnsupportedOperationException {
   throw new UnsupportedOperationException(); }

public void insert( T newItem ) { root = insertItem( root, newItem ); }

public T retrieve( KT searchKey ) { return retrieveItem( root, searchKey ); }

public void delete( KT searchKey ) throws TreeException {
   root = deleteItem( root, searchKey ); }

public void delete( T item ) throws TreeException {
   root = deleteItem( root, item.getKey() ); }
```

BST - REFERENCE-BASED IMPLEMENTATION 5

BINARY SEARCH TREE C

```
// Internal method: insertItem.
protected TreeNode<T> insertItem( TreeNode<T> tNode, T newItem ) {
   TreeNode<T> newSubtree;
   if( tNode == null ) { // Position of insertion found.
      tNode = new TreeNode<T>( newItem, null, null ); // Create a new node.
      return tNode; } // Insert new node after leaf.
   T nodeItem = tNode.item;
   // Search for the insertion position.
   if( newItem.getKey().compareTo( nodeItem.getKey() ) < 0 ) { // Search left subtree.
      newSubtree = insertItem( tNode.leftChild, newItem );
      tNode.leftChild = newSubtree;
      return tNode; }
   else { // Search right subtree.
      newSubtree = insertItem( tNode.rightChild, newItem );
      tNode.rightChild = newSubtree;
      return tNode; }
}
```

BST - REFERENCE-BASED IMPLEMENTATION 6

BINARY SEARCH TREE D

```
// Internal method: retrieveItem.
protected T retrieveItem( TreeNode<T> tNode, KT searchKey ) {
   T treeItem;
   if( tNode == null ) { treeItem = null; }
   else {
      T nodeItem = tNode.item;
      if( searchKey.compareTo( nodeItem.getKey() ) == 0 ) { // Item is in the root.
         treeItem = tNode.item; }
      else if( searchKey.compareTo( nodeItem.getKey() ) < 0 ) { // Search left tree.
         treeItem = retrieveItem( tNode.leftChild, searchKey ); }
      else { treeItem = retrieveItem( tNode.rightChild, searchKey ); } }
   return treeItem;
}
```

BST - REFERENCE-BASED IMPLEMENTATION 7

BINARY SEARCH TREE E

```
// Internal method: deleteItem.
protected TreeNode<T> deleteItem( TreeNode<T> tNode, KT searchKey ) {
   TreeNode<T> newSubtree;
   if( tNode == null ) { throw new TreeException( "TreeException: key not found!" ); }
   else {
      T nodeItem = tNode.item;
      if( searchKey.compareTo( nodeItem.getKey() ) == 0 ) { // Item is in the root.
         tNode = deleteNode( tNode ); }
      else if( searchKey.compareTo( nodeItem.getKey() ) < 0 ) { // Search left tree.
         newSubtree = deleteItem( tNode.leftChild, searchKey );
         tNode.leftChild = newSubtree; }
      else {
         newSubtree = deleteItem( tNode.rightChild, searchKey );
         tNode.rightChild = newSubtree; } }
   return tNode;
}
```

BST - REFERENCE-BASED IMPLEMENTATION 8

BINARY SEARCH TREE F

```
// Internal method: deleteNode.
protected TreeNode<T> deleteNode( TreeNode<T> tNode ) {
   // 4 cases to consider: tNode is a leaf (1); tNode has no left child (2);
   //                      tNode has no right child (3); tNode has 2 children (4).
   T replacementItem;
   if( ( tNode.leftChild == null ) && ( tNode.rightChild == null ) ) { // Case (1).
      return null; }
   else if( tNode.leftChild == null ) { return tNode.rightChild; } // Case (2).
   else if( tNode.rightChild == null ) { return tNode.leftChild; } // Case (3).
   else { // Case (4): retrieve and delete the inorder successor.
      replacementItem = findLeftmost( tNode.rightChild );
      tNode.item = replacementItem;
      tNode.rightChild = deleteLeftmost( tNode.rightChild );
      return tNode; }
}
```

BST - REFERENCE-BASED IMPLEMENTATION 9

BINARY SEARCH TREE G

```
   // Internal method: findLeftmost.
   protected T findLeftmost( TreeNode<T> tNode ) {
      if( tNode.leftChild == null ) { return tNode.item; }
      else { return findLeftmost( tNode.leftChild ); }
   }

   // Internal method: deleteLeftmost.
   protected TreeNode<T> deleteLeftmost( TreeNode<T> tNode ) {
      if( tNode.leftChild == null ) { return tNode.rightChild; }
      else {
         tNode.leftChild = deleteLeftmost( tNode.leftChild );
         return tNode; }
   }

}
```

BST - REFERENCE-BASED IMPLEMENTATION 10

SEARCH TREE ITEM

```java
package Tree;

// Class to store a record of fields (search key included) in a binary search tree.
public class SearchTreeItem< T, KT extends Comparable<? super KT > > extends
KeyedItem<KT> {

   public T data; // Data field.

   // Constructors.
   public SearchTreeItem( KT k ) { super(k); data = null; }
   public SearchTreeItem( T d, KT k ) { super(k); data = d; }

}
```

BST - REFERENCE-BASED IMPLEMENTATION 11

SEARCH TREE TEST A

```java
import Tree.BinarySearchTree;
import Tree.SearchTreeItem;
import Tree.TreeIterator;
import java.lang.String;
import java.lang.Integer;

public class SearchTreeTest {

   public static void main( String[] args ) {
      // Create an item and a new binary search tree with that item as root.
      SearchTreeItem< Integer, String > rootItem =
         new SearchTreeItem< Integer, String >( 0, "Janet" );
      BinarySearchTree< SearchTreeItem< Integer, String >, String > bst =
         new BinarySearchTree< SearchTreeItem< Integer, String >, String >( rootItem );
```

BST - REFERENCE-BASED IMPLEMENTATION 12

SEARCH TREE TEST B

```
// Create and insert 6 new items in the binary search tree: from i1 to i6.
SearchTreeItem<Integer,String> i1 = new SearchTreeItem<Integer,String>(1,"Bob");
bst.insert( i1 );
SearchTreeItem<Integer,String> i2 = new SearchTreeItem<Integer,String>(2,"Tom");
bst.insert( i2 );
SearchTreeItem<Integer,String> i3 = new SearchTreeItem<Integer,String>(3,"Alan");
bst.insert( i3 );
SearchTreeItem<Integer,String> i4 = new SearchTreeItem<Integer,String>(4,"Ellen");
bst.insert( i4 );
SearchTreeItem<Integer,String> i5 = new SearchTreeItem<Integer,String>(5,"Karen");
bst.insert( i5 );
SearchTreeItem<Integer,String> i6 = new SearchTreeItem<Integer,String>(6,"Wendy");
bst.insert( i6 );
// Delete an item in the binary search tree using a search key.
bst.delete( "Janet" );
```

BST - REFERENCE-BASED IMPLEMENTATION 13

SEARCH TREE TEST C

```
      // Test the binary tree iterator on the binary search tree.
      TreeIterator< SearchTreeItem< Integer, String > > bstIter =
         new TreeIterator< SearchTreeItem< Integer, String > >( bst );
      bstIter.setInorder();
      System.out.println( "---" );
      while( bstIter.hasNext() ) {
         SearchTreeItem< Integer, String > currItem = bstIter.next();
         System.out.println( currItem.getKey() + ":    " + currItem.data ); }
      System.out.println( "---" );
   }

}
```

BST - EFFICIENCY 1

To evaluate the efficiency of a binary searh tree **consider the relationship between its height and the visits (comparisons)** you need to perform an operation.

Each operation requires a number of comparisons equal to the number of nodes along the path traveled to perform the operation.

The max number of comparisons for retrieval, insertion, or deletion is the height of the tree.

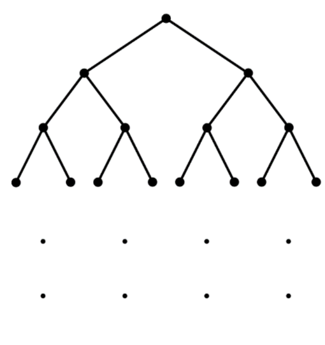
What are the max and min heights of a binary search tree with n nodes ?

BST - EFFICIENCY 2

Theorem: A full binary tree of height **h >= 0** has $\mathbf{2^h-1}$ nodes.

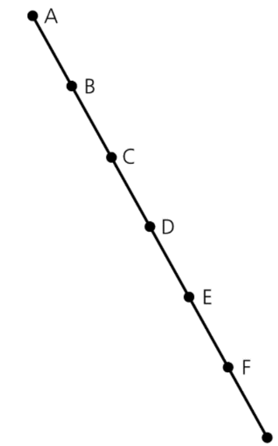
Theorem: The max number of nodes that a binary tree of height **h** can have is $\mathbf{2^h-1}$.

Theorem: The min height of a binary tree with **n** nodes is **ceiling($\mathbf{\log_2}$(n+1))**.



Level	Number of nodes at this level	Number of nodes at this and previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
.	.	.
.	.	.
.	.	.
h	2^{h-1}	$2^h - 1$

BST - EFFICIENCY 3



MAXIMUM HEIGHT OF A BINARY TREE

A binary tree with **n** nodes has

maximum height = n

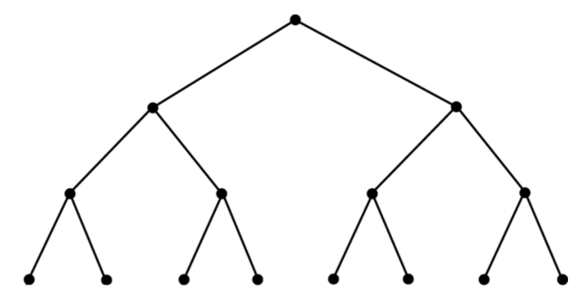
(when the tree structure is a continuous chain of nodes).

MINIMUM HEIGHT OF A BINARY TREE

A binary tree with **n** nodes has

minimum height = ceiling($\log_2$(n+1))

(when the tree is perfectly balanced).



BST - EFFICIENCY 4

The height of a particular BST depends on the order in which insertion/deletion operations are performed.

Table: The order of the retrieval, insertion, deletion, and traversal operations for the reference-based implementation of the ADT binary search tree.

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

Note: If a BST is complete, the time it takes to search it for a value is about the same as that required for a binary search of an array. However, as you go from **balanced trees (min height)** toward **extremely unbalanced trees** (i.e. with a linear structure, **max height**), the height approaches **n**: the number of nodes. This number equal the max number of comparisons needed when searching a linked list of **n** nodes.

BST - TREESORT SORTING ALGORITHM

THE TREESORT ALGORITHM

The **treesort** algorithm uses the ADT binary search tree to sort an array of records into search-key order. Its efficiency for an array with **n** items is:

- in the average case: $\mathbf{O(n \cdot \log_2(n))}$; and
- in the worst case: $\mathbf{O(n^2)}$.

```
// Treesort algorithm (pseudocode). Sorts n integers in input array into ascending order.
public void treesort( ArrayType anArray, int n ) {
   // A: Inserts all the array elements into a binary search tree.
   for( int i = 0; i < n; i++ ) { bst.insertItem( anArray[i] ); }
   // B: Traverse the binary search tree using the inorder traversal.
   //    As you visit a tree node, sequentially copy its data into the input array.
   TreeIterator iter = new TreeIterator( bst ); iter.setInorder(); int j = 0;
   while( iter.hasNext() ) {
      int curr = iter.next();
      anArray[j] = curr;
      j++; } }
```

BST - SAVING INTO FILE 1

You can save a binary search tree by adding the **java.io.Serializable** interface to the various classes in the implementation of the binary search tree.
Otherwise, the following are 2 algorithms for saving and restoring a BST:

Saving a binary search tree and then restoring it to its original shape:

1. first, uses **preorder** traversal to save the tree to a file;
2. then, uses inserts in the same sequence to re-create the original tree.

Saving a binary tree and then restoring it to a balanced shape:

1. uses **inorder** traversal to save the tree to a file,
2. exploit the sorted data to properly perform insertions to max balancing.

Note: Can be accomplished only if data is sorted, and number of nodes is known.

See: docs.oracle.com/javase/8/docs/api/java/io/serializable

BST - SAVING INTO FILE 2

SAVING A BINARY TREE AND THEN RESTORING IT TO ITS ORIGINAL SHAPE A

1. first, uses **preorder** traversal to save the tree to a file;
2. then, uses inserts in the same sequence to re-create the original tree.

```
// SAVE BINARY SEARCH TREE INTO FILE - PREORDER TRAVERSAL
TreeIterator< SearchTreeItem< Integer, String > > saveIter1 =
   new TreeIterator< SearchTreeItem< Integer, String > >( bst ); // Init the iterator.
saveIter1.setPreorder();
try { // Open file for writing.
   PrintWriter writer = new PrintWriter( "bst__preorder.txt", "UTF-8" );
   while( saveIter1.hasNext() ) { // Binary search tree traversal using iterator.
      SearchTreeItem< Integer, String > currItem = saveIter1.next();
      writer.println( currItem.getKey() + ": " + currItem.data ); }
   writer.close(); } // Close file.
catch( FileNotFoundException e ) {}
catch( UnsupportedEncodingException e ) {}
```

BST - SAVING INTO FILE 3

SAVING A BINARY TREE AND THEN RESTORING IT TO ITS ORIGINAL SHAPE B

1. first, uses **preorder** traversal to save the tree to a file;
2. then, uses inserts in the same sequence to re-create the original tree.

```
// LOAD BINARY SEARCH TREE FROM FILE – SEQUENTIAL INSERTION
BinarySearchTree< SearchTreeItem< Integer, String >, String > bst2 =
   new BinarySearchTree< SearchTreeItem< Integer, String >, String >();
try {
   Scanner scannerFile = new Scanner( new File( "bst__preorder.txt" ) );
   while( scannerFile.hasNext() ) {
      Scanner scannerString = new Scanner( scannerFile.nextLine() );
      scannerString.useDelimiter( ": " );
      String currKey = scannerString.next();
      Integer currData = Integer.parseInt( scannerString.next() );
      bst2.insert( new SearchTreeItem< Integer, String >( currData, currKey ) ); }
   scannerFile.close(); }
catch( FileNotFoundException exc ) {}
```

BST - SAVING INTO FILE 4

SAVING A BINARY TREE AND THEN RESTORING IT TO A BALANCED SHAPE A

1. uses **inorder** traversal to save the tree to a file,
2. exploit the sorted data to properly perform insertions to max balancing.

```
// SAVE BINARY SEARCH TREE INTO FILE - INORDER TRAVERSAL
TreeIterator< SearchTreeItem< Integer, String > > saveIter2 =
   new TreeIterator< SearchTreeItem< Integer, String > >( bst ); // Init the iterator.
saveIter2.setInorder();
try { // Open file for writing.
   PrintWriter writer = new PrintWriter( "bst__inorder.txt", "UTF-8" );
   while( saveIter2.hasNext() ) { // Binary search tree traversal using iterator.
      SearchTreeItem< Integer, String > currItem = saveIter2.next();
      writer.println( currItem.getKey() + ": " + currItem.data ); }
   writer.close(); } // Close file.
catch( FileNotFoundException e ) {}
catch( UnsupportedEncodingException e ) {}
```

BST - SAVING INTO FILE 5

SAVING A BINARY TREE AND THEN RESTORING IT TO A BALANCED SHAPE B

1. uses **inorder** traversal to save the tree to a file,
2. exploit the sorted data to properly perform insertions to max balancing.

```
// Pseudocode of the algorithm to restore a binary search tree to a balanced shape.
// Builds a min-height binary search tree from n sorted values in a file, returns root.
public TreeNode readTree( FileType inputFile, int n ) {
   TreeNode treeNode = new TreeNode(); // Create a new empty tree node.
   if( n > 0 ) {
      treeNode.leftChild = readTree ( inputFile, n/2 ); // Build the left subtree.
      treeNode.item = read the root data from file. // Set the root item.
      treeNode.rightChild = readTree( inputFile, (n-1)/2 ); } // Build the left subtree.
   return treeNode;
}
```

BST - SAVING INTO FILE 6

SAVING A BINARY TREE AND THEN RESTORING IT TO A BALANCED SHAPE C

1. uses **inorder** traversal to save the tree to a file,
2. exploit the sorted data to properly perform insertions to max balancing.

```
// LOAD BINARY SEARCH TREE FROM FILE – SEQUENTIAL INSERTION (1)
BinarySearchTree< SearchTreeItem< Integer, String >, String > bst3 = ...
LinkedList< SearchTreeItem< Integer, String > > list = ...
try { Scanner scannerFile = new Scanner( new File( "bst__inorder.txt" ) );
      while( scannerFile.hasNext() ) {
         Scanner scannerString = new Scanner( scannerFile.nextLine() );
         scannerString.useDelimiter( ": " );
         String currKey = scannerString.next();
         Integer currData = Integer.parseInt( scannerString.next() );
         list.add( new SearchTreeItem< Integer, String >( currData, currKey ) ); }
      scannerFile.close(); }
catch( FileNotFoundException exc ) {}
bst3.loadInorder( list );
```

BST - SAVING INTO FILE 7

SAVING A BINARY TREE AND THEN RESTORING IT TO A BALANCED SHAPE D

1. uses **inorder** traversal to save the tree to a file,
2. exploit the sorted data to properly perform insertions to max balancing.

```
// LOAD BINARY SEARCH TREE FROM FILE - SEQUENTIAL INSERTION (2)
public void loadInorder( LinkedList<T> f ) { root = readTree( f, 0, f.size()-1 ); }

private TreeNode< T > readTree( LinkedList<T> f, int min, int max ) {
   TreeNode< T > treeNode = null;
   if( max >= min ) {
      int rootIdx = (min+max)/2; int leftMax = rootIdx-1; int rightMin = rootIdx+1;
      treeNode = new TreeNode< T >( null );
      treeNode.leftChild = readTree( f, min, leftMax ); // Build left subtree.
      treeNode.item = f.get( rootIdx ); // Get current root data from file.
      treeNode.rightChild = readTree( f, rightMin, max ); } // Build right subtree.
   return treeNode;
}
```

BST - BINARY SEARCH TREES IN THE JCF 1

The JCF has 2 **binarySearch** methods (see **java.util.Collections**):

1. based on the natural ordering of elements:
   ```
   static <T> int binarySearch( List<? extends Comparable<? super T>> l, T k );
   ```

2. based on a specified **Comparator** object:
   ```
   static <T> int binarySearch( List<? extends T> l, T k, Comparator<? super T> c );
   ```

Both assume list in ascending order. If element is found, its index is returned. If element not found, a negative value **val** is returned. **val** can be used to insert the element in the sorted list (**insertIndex = -val-1**), even at the end.

See: docs.oracle.com/javase/8/docs/api/java/util/collections
See: docs.oracle.com/javase/8/docs/api/java/util/comparator

BST - BINARY SEARCH TREES IN THE JCF 2

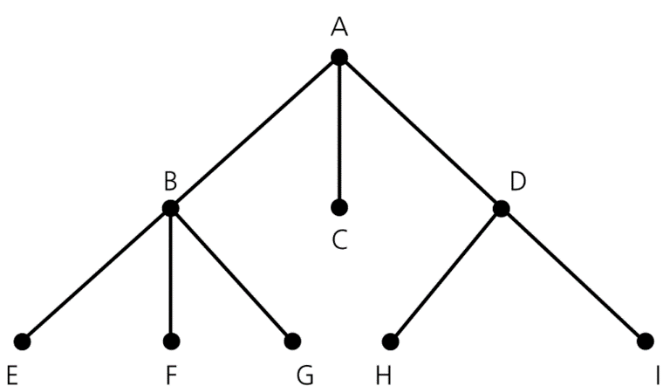
```
// Example of usage of the JCF binary search algorithm
public class JCFSearchExample {
   public static void main( String[] args ) {
      String[] names = {"Janet","Michael","Pat","Craig","Andrew","Sarah","Evan","Anita"};
      LinkedList<String> nameList = new LinkedList<String>();
      nameList.addAll( Arrays.asList( names ) );
      System.out.println( nameList );
      Collections.sort( nameList );
      System.out.println( nameList );
      String name = "Maite";
      int loc = Collections.binarySearch( nameList, name );
      if( loc < 0 ) {
         System.out.println( name + " should be inserted at index " + -(loc+1) );
         nameList.add( -(loc+1), name );
         System.out.println( nameList ); }
      else { System.out.println( name + " was found in location " + loc ); }
   }
}
```

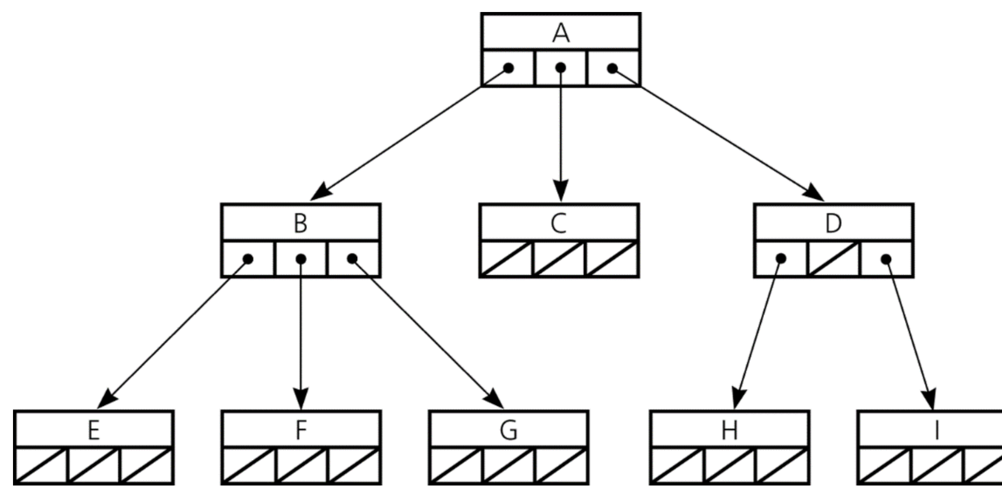
N-ARY TREES

AN N-ARY TREE: DEFINITION

An n-ary tree is a generalization of a binary tree whose nodes each can have no more than **n** children.

Example: In figure, a 3-ary tree (**n=3**) and its reference-based implementation.





Kettering
UNIVERSITY