Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University
Kazan, Russia

Trees

A Question!

- Last time, we learned that a hash table can insert and retrieve elements in O(1)
- This is as fast as it can be!
- Why do we need Trees then?
- Stay patient, we will discuss this after we are done with Binary Search Trees

Topic Overview

- Trees
- Oriented Trees
- Ordered Trees
- Binary Trees
- Tree Traversal Algorithms
- Implementation

Tree

- A tree combines the advantages of two other data structures:
  - An ordered array
  - A linked list

Ordered Array

- Quick to search for a particular element, using binary search
- On the other hand, insertions are slow
  - First need to find where the object will go
  - Then move all the objects with greater keys up one space to make the room

Linked List

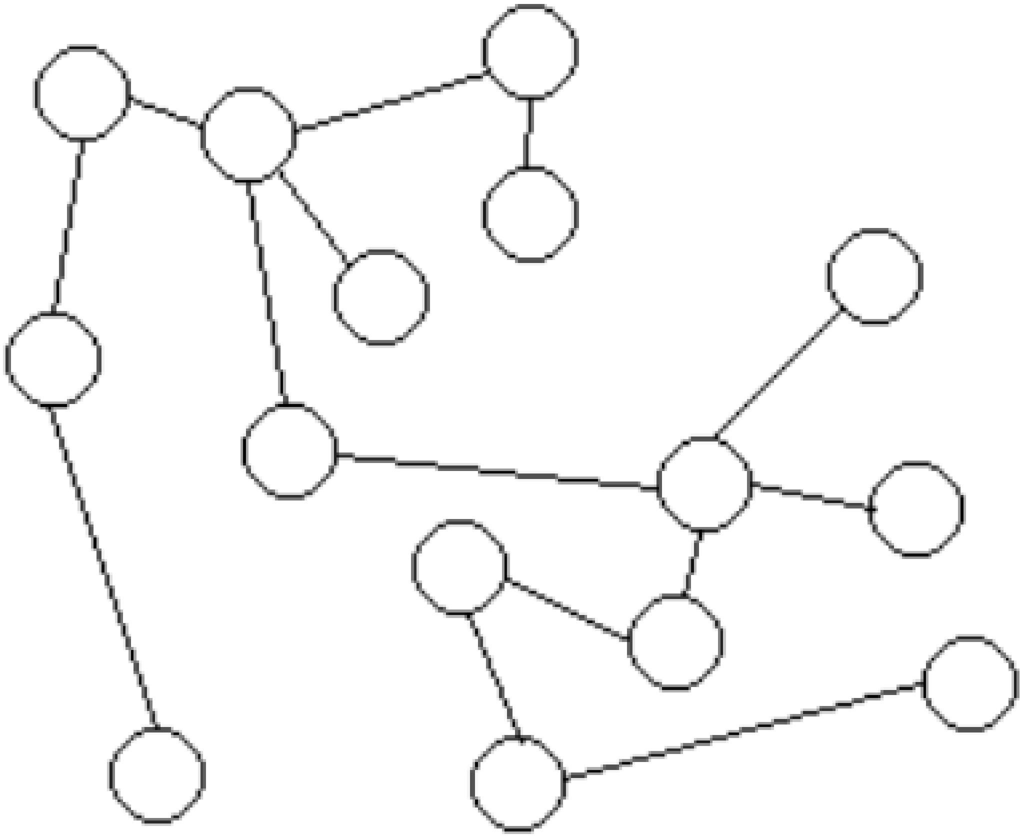
- Insertions and deletions are quick
  - Simply requires changing a few constant number of references
- On the other hand, finding a particular element is slow
  - Must start at the beginning of the linked list
  - Visit each element until you find the one you're looking for.
- What if we made the linked list ordered? Will it help?

Trees to Rescue

- It would be nice to have a data structure
  - With quick insertions and deletions of a linked list
  - And, the quick searching of an ordered array
- Trees provide both these characteristics
- Our main focus will be a binary tree
- But let’s first start discussing trees in general

Tree

- Consists of nodes connected by edges



Node

(a.k.a. vertex) a data element in the tree

Edge

(a.k.a. branch, or link) a connection between two nodes

Empty tree

has zero nodes

Size

the size of a tree is the number of nodes

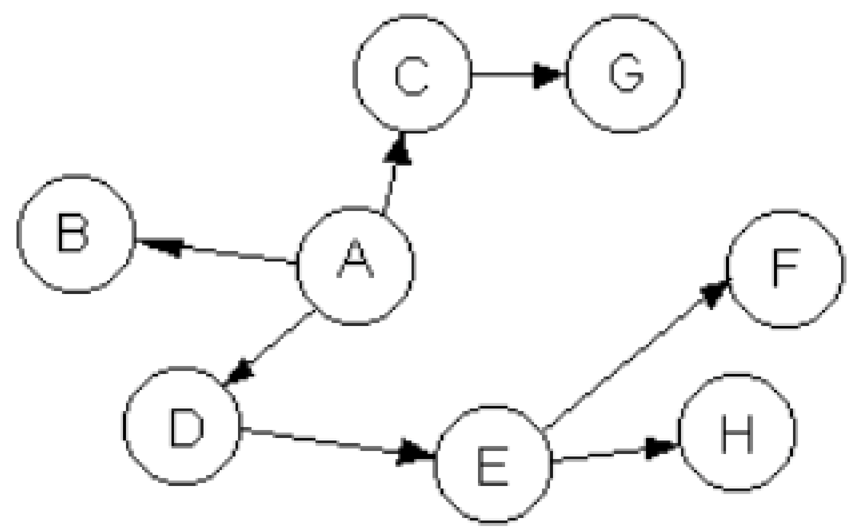
Path

a sequence $n_0e_1n_1e_2n_2...e_kn_k$ where $k \geq 0$ and e_i connectes n_{i-1} and n_i.

Oriented Trees

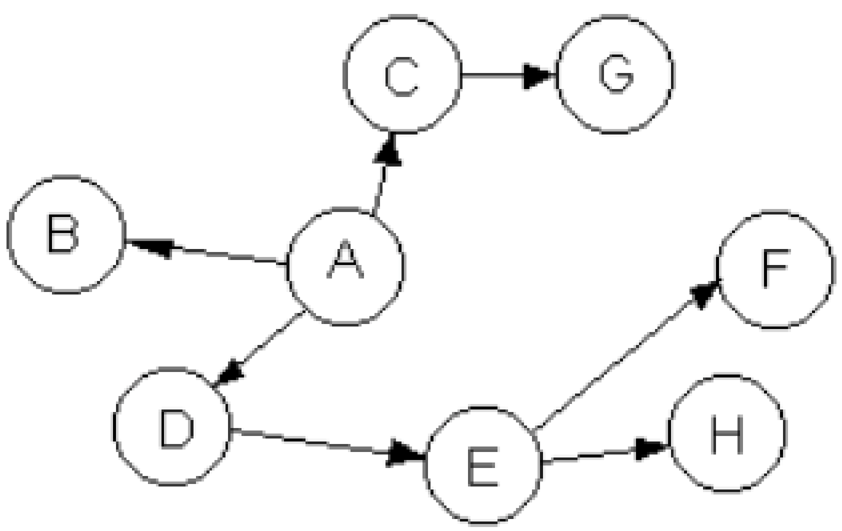
Oriented Trees

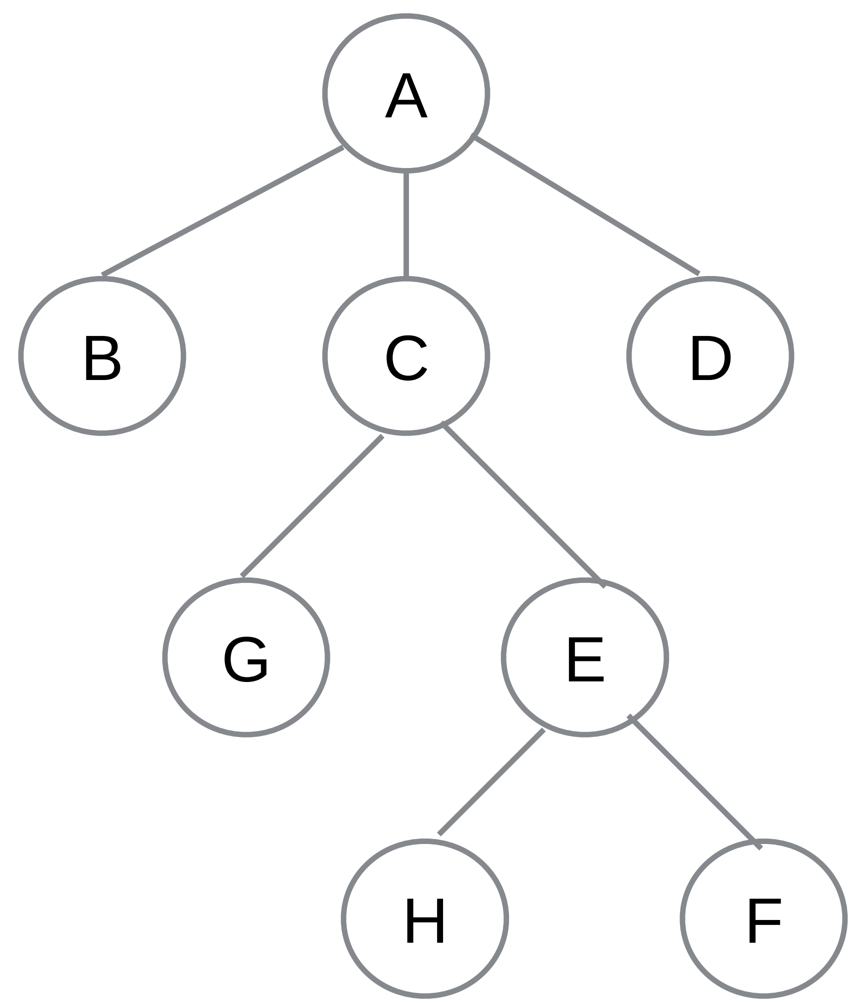
- A tree used to represent a hierarchical data.
- All edges are directed outward from a distinguished node called the root node



Oriented Trees

- Usually drawn with the root at the top, all edges pointing downward, the arrows are thus redundant and often omitted.





Definitions

Parent

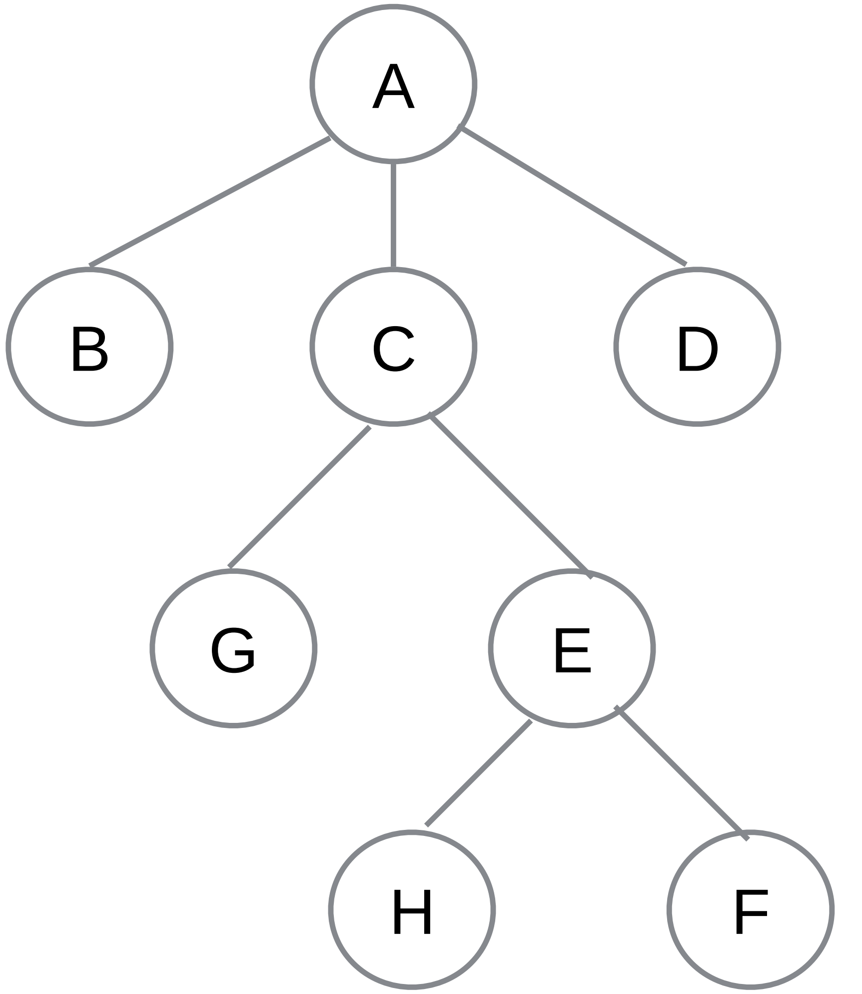
A is the parent of B and C and D

Children

The children of A are B and C and D

Siblings

B and C are siblings



Definitions

Root node

The only node without a parent

Leaf node

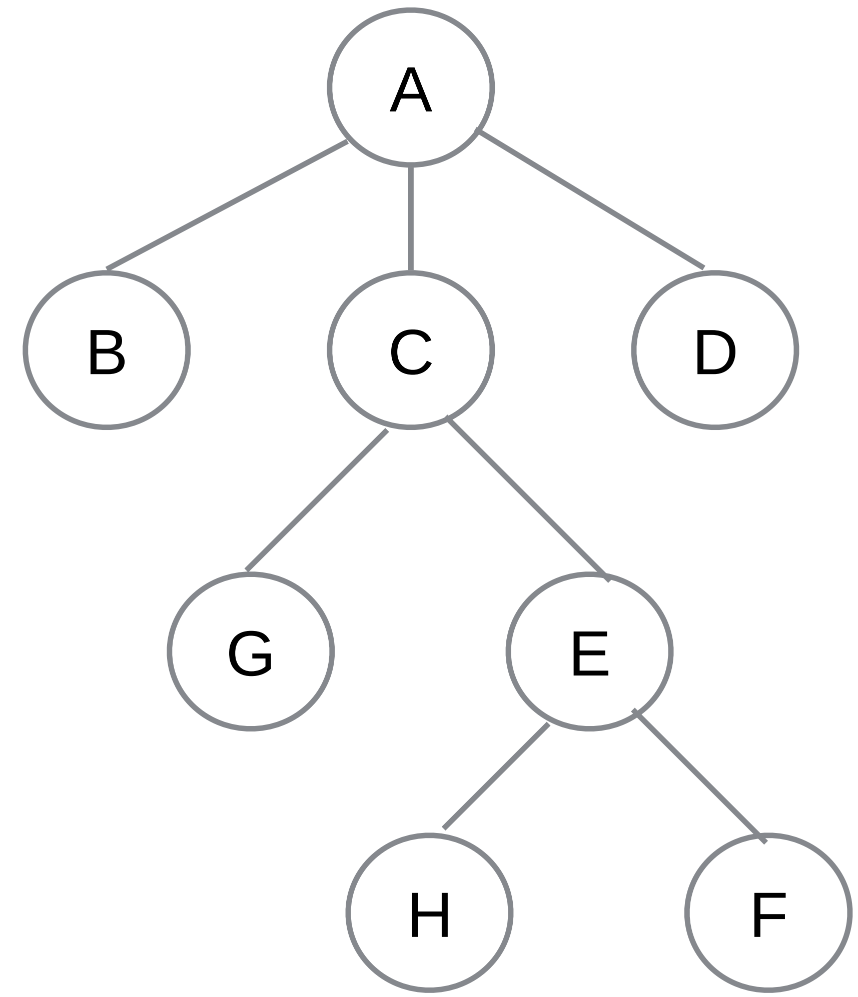
A node without children

Degree(n)

The number of children of n

Degree(t)

The greatest degree of the nodes in t



Definitions

Level(n)

Level(n) = if n is the root then 0 else 1 + Level(Parent(n))

Depth(n)

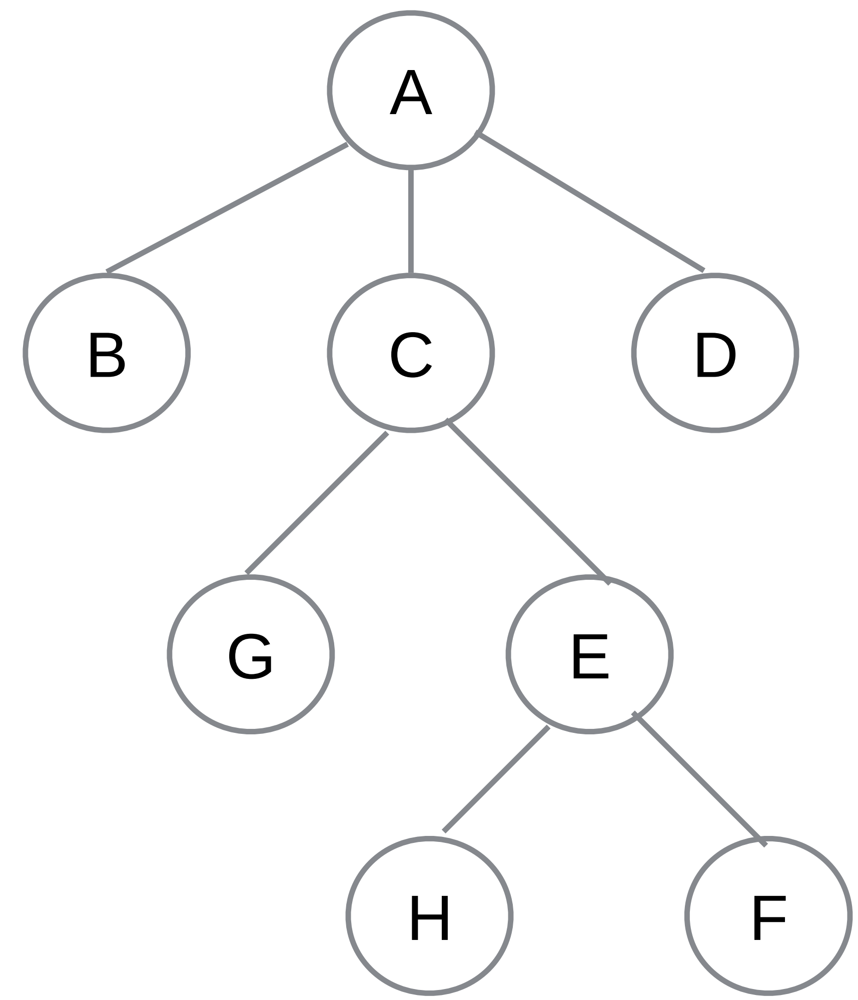
Same as Level(n)

Height(n)

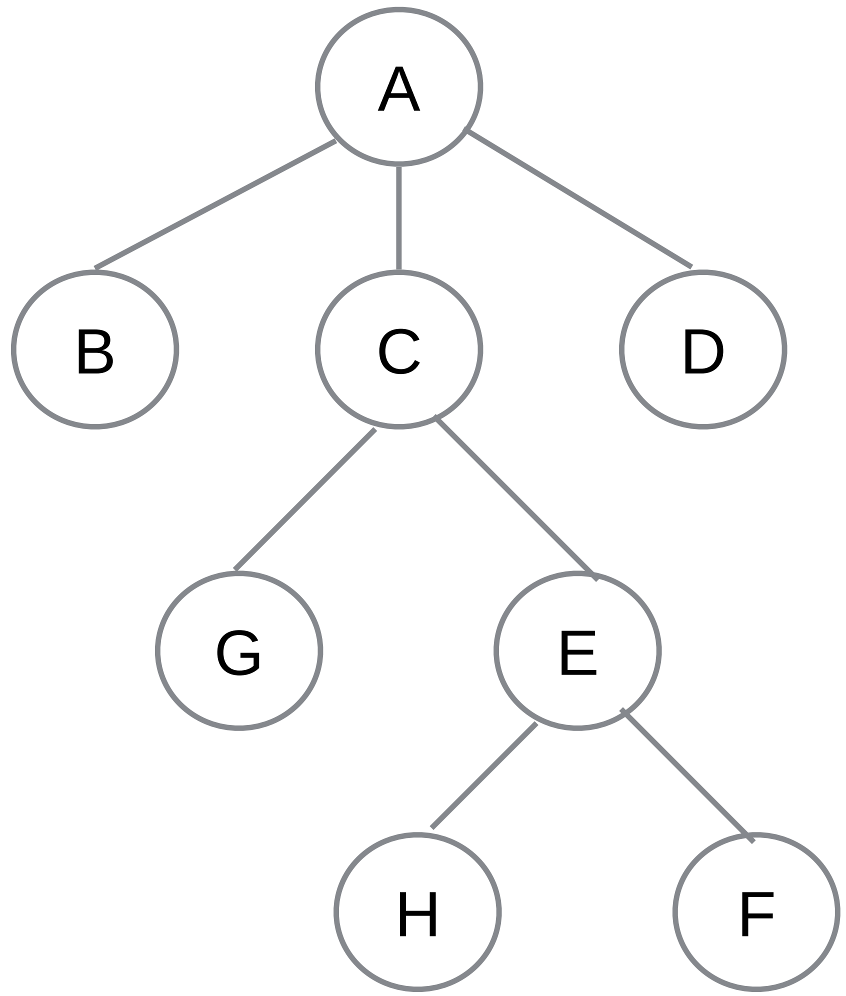
Maximum length over all paths from n to a leaf

Height(t)

The height of t's root node



Definitions



Ancestors(n)

Ancestors(n) = if n is the root then { } else {Parent(n)} union Ancestors(Parent(n))

Descendants(n)

The set of all nodes reachable from n following the edges leaving it in the direction of the arrows

Definitions

PathLength(t)

The sum of all the depths of all the nodes in t

Empty Tree

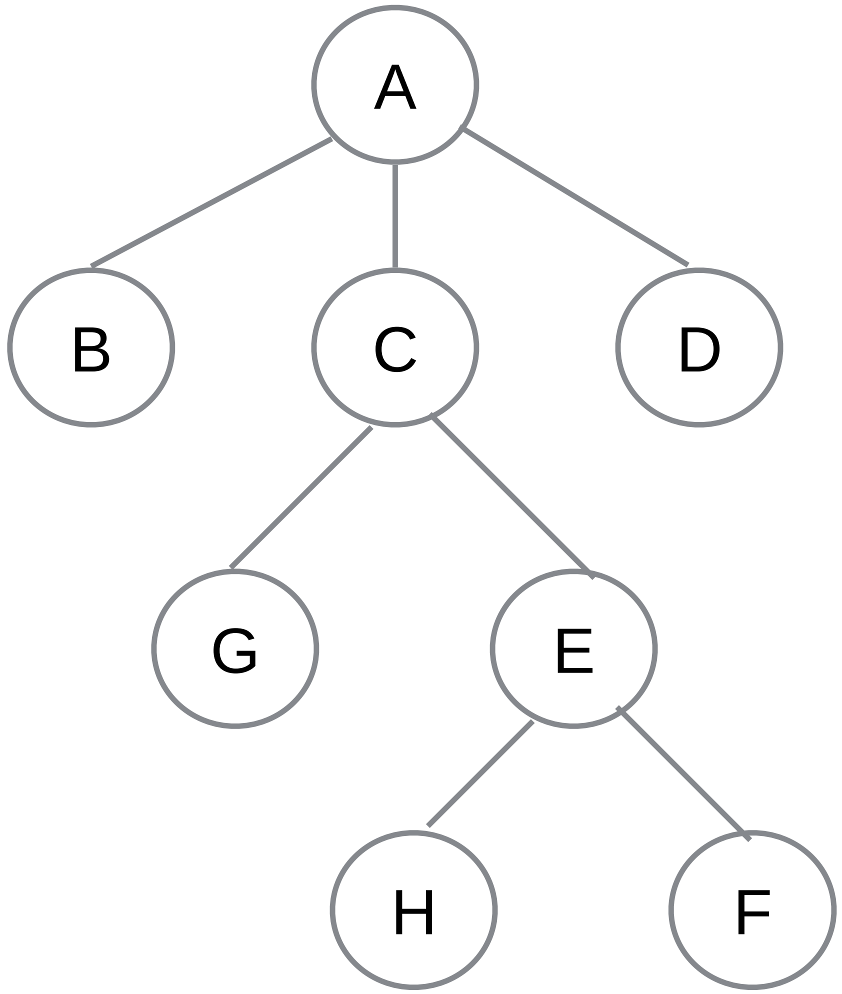
A tree with zero nodes

Singleton Tree

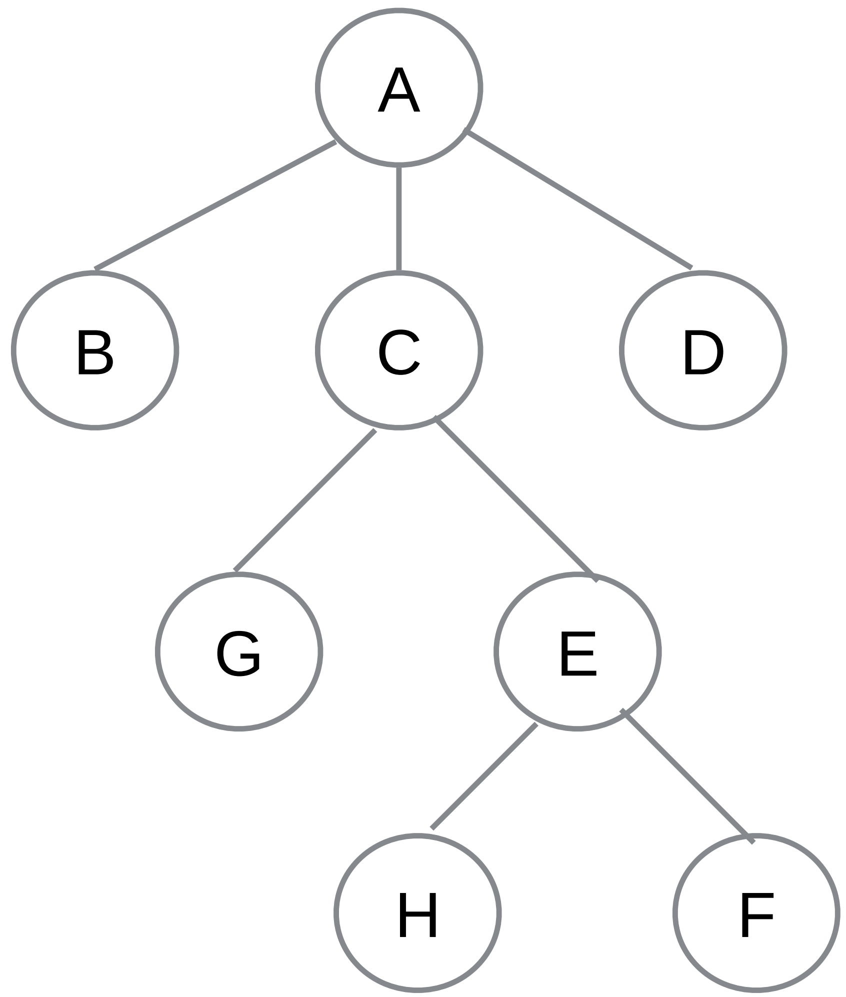
A tree with only one node

Subtree(n)

The subtree rooted at n



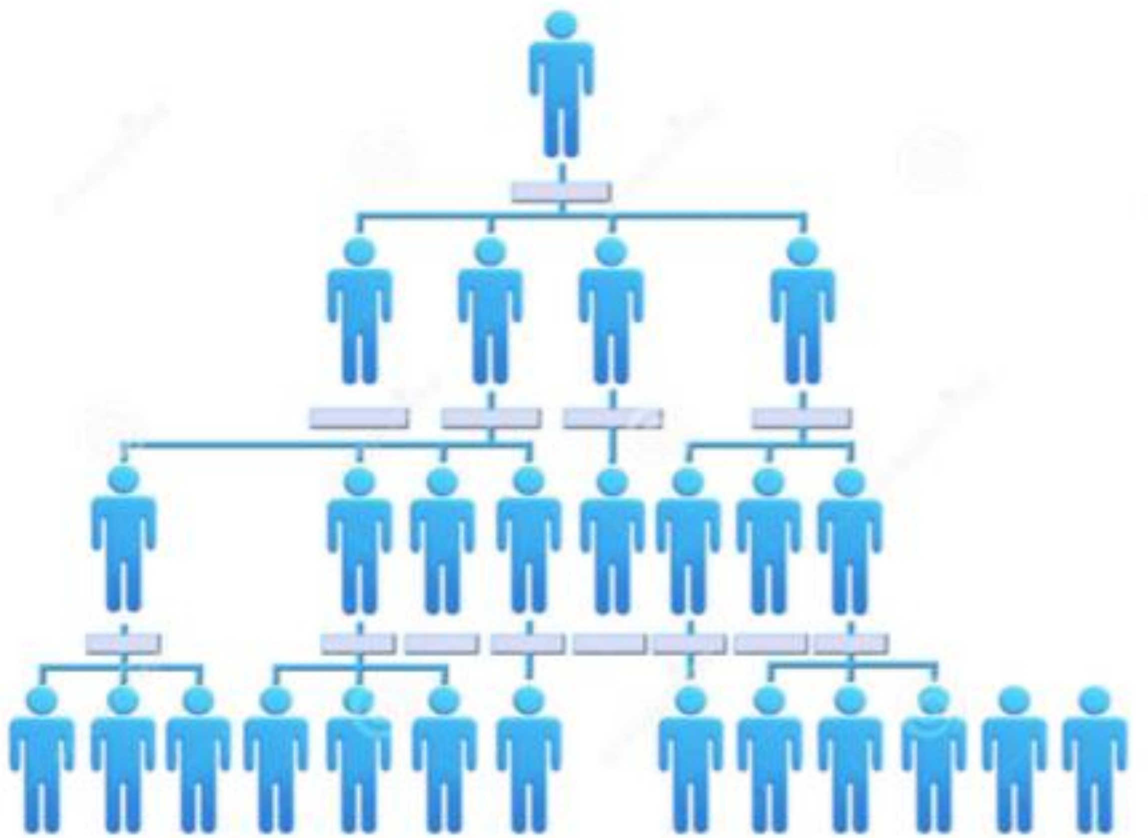
Definitions

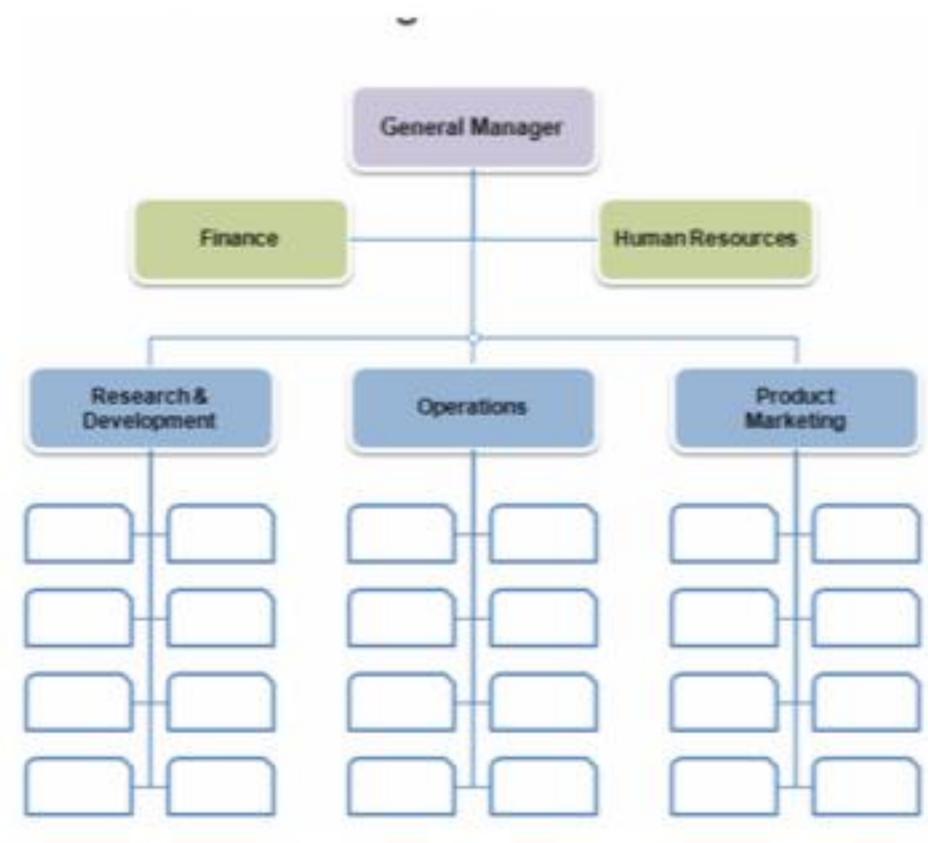


k-node

A node with k children. A 0-node is a leaf. A 1-node has exactly one child. A 2-node has exactly two children. Etc.

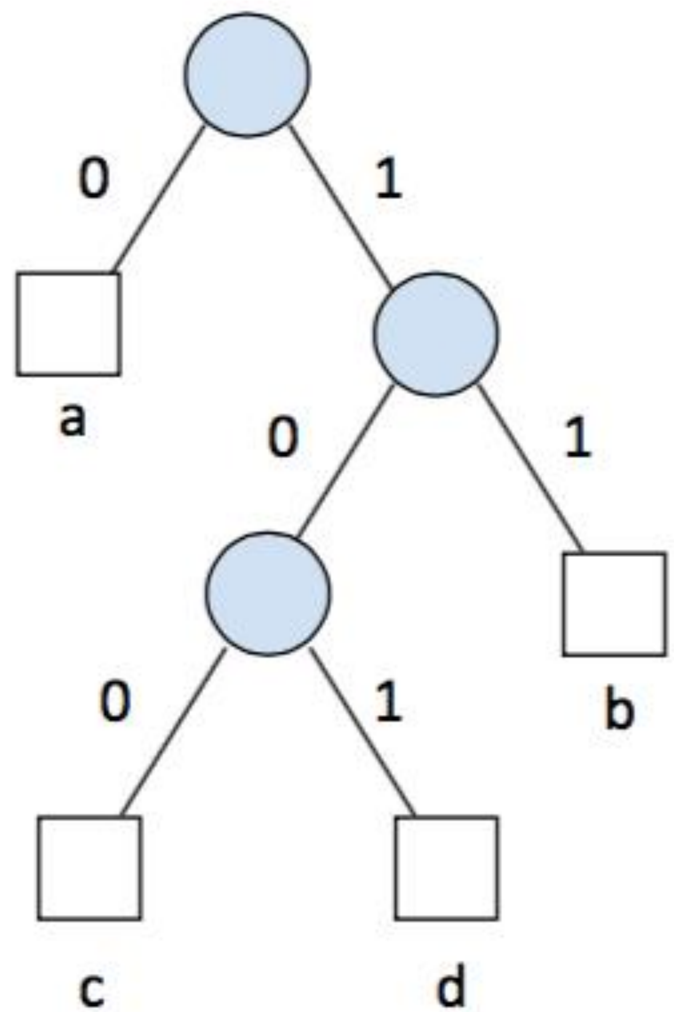
Uses of Trees





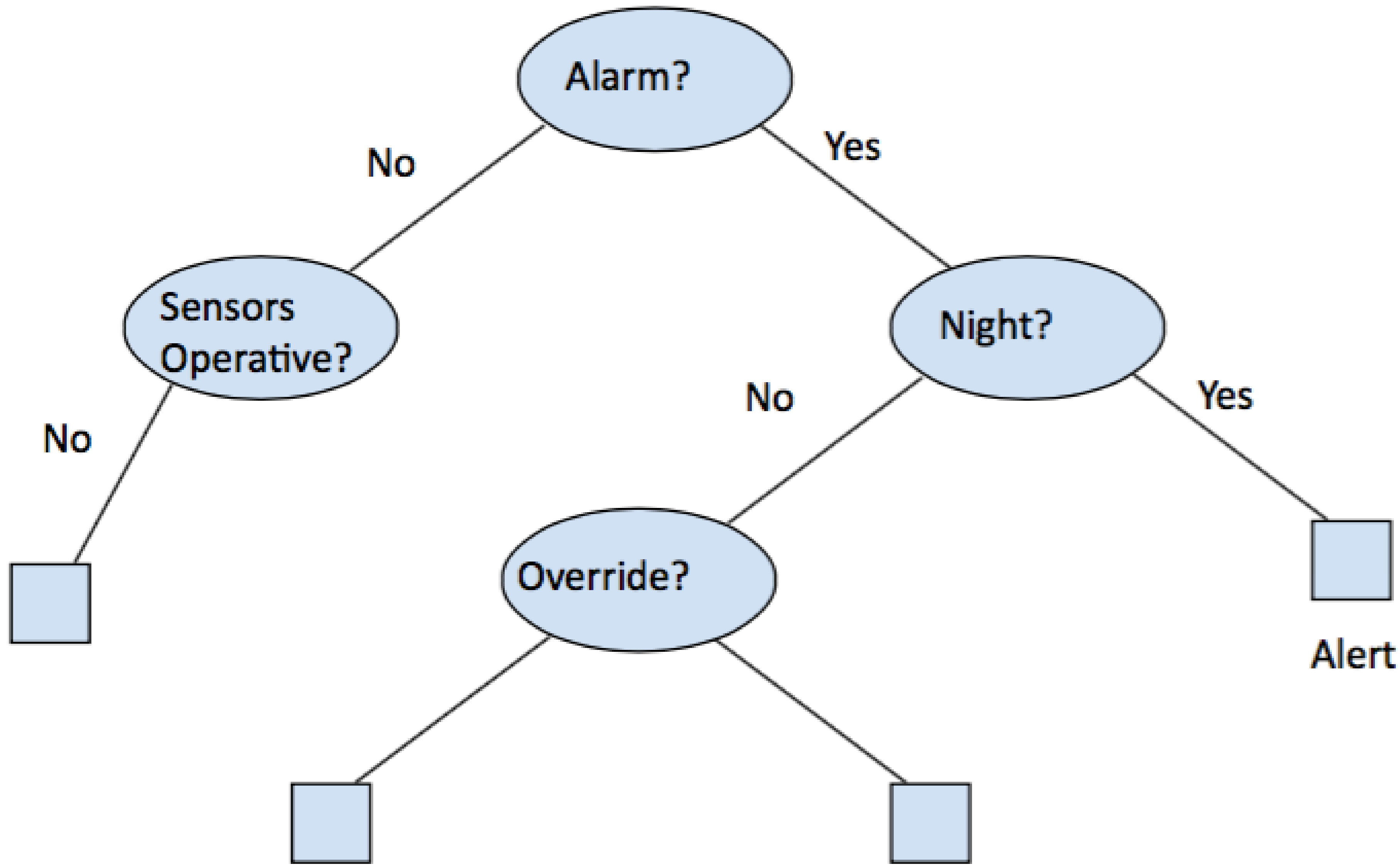
Organization Tree

Uses of Trees



Code Tree

Uses of Trees

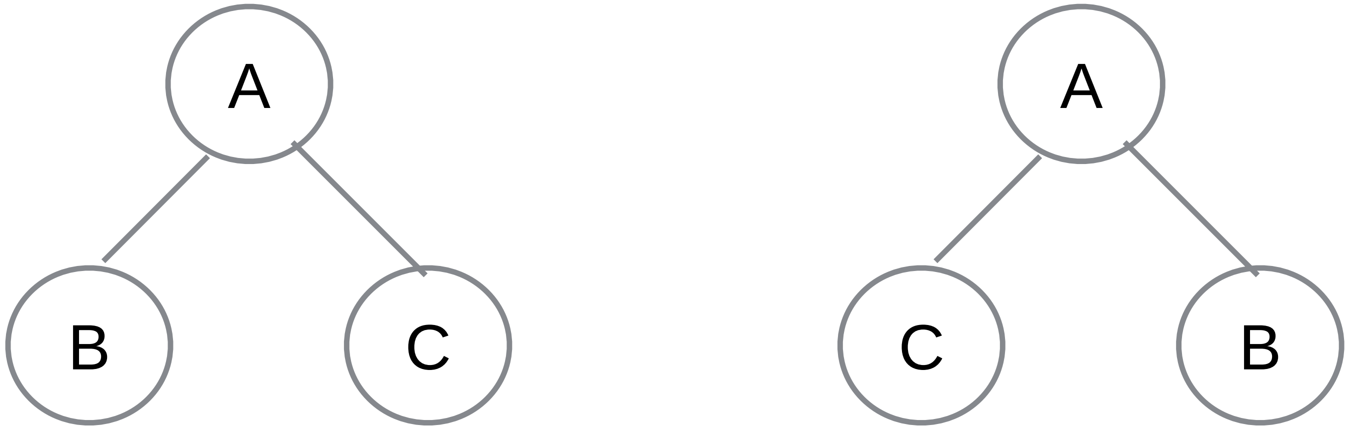


Decision Trees

Ordered Trees

Ordered Trees

- An oriented tree in which the children of a node are somehow **ordered**.



T1

T2

If T1 and T2 are ordered trees then T1 =! T2, otherwise T1 = T2

Types of Ordered Trees

- Fibonacci Tree
- Binomial tree
- k-ary Tree

Fibonacci Trees

- A Fibonacci Tree (F_k) is defined by

  - F_0 is the empty tree
  - F_1 is a tree with only one node
  - F_{k+2} is a node whose left subtree is a F_{k+1} tree and whose right subtree is a F_k tree.

Exercise: Draw the trees F_0 through F_5.

Binomial Trees

- The Binomial Tree (B_k) consists of a node with k children. The first child is the root of a B_{k-1} tree, the second is the root of a B_{k-2} tree, etc.

Exercise: Draw the trees B_0 through B_5.

k-ary Trees

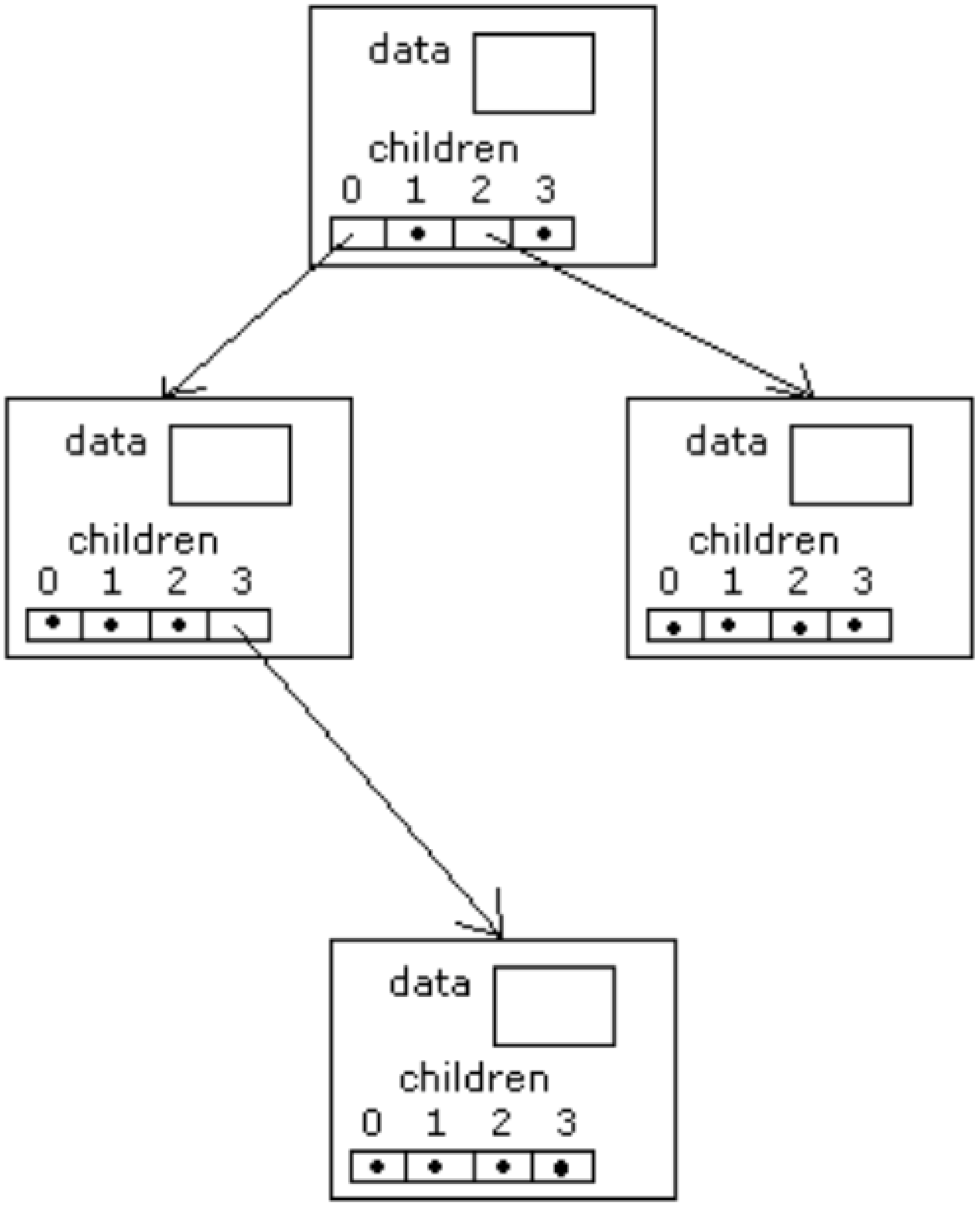
- A tree in which the children of a node appear at distinct index positions in 0..k-1
- Maximum number of children for a node is k

Types of k-ary Trees

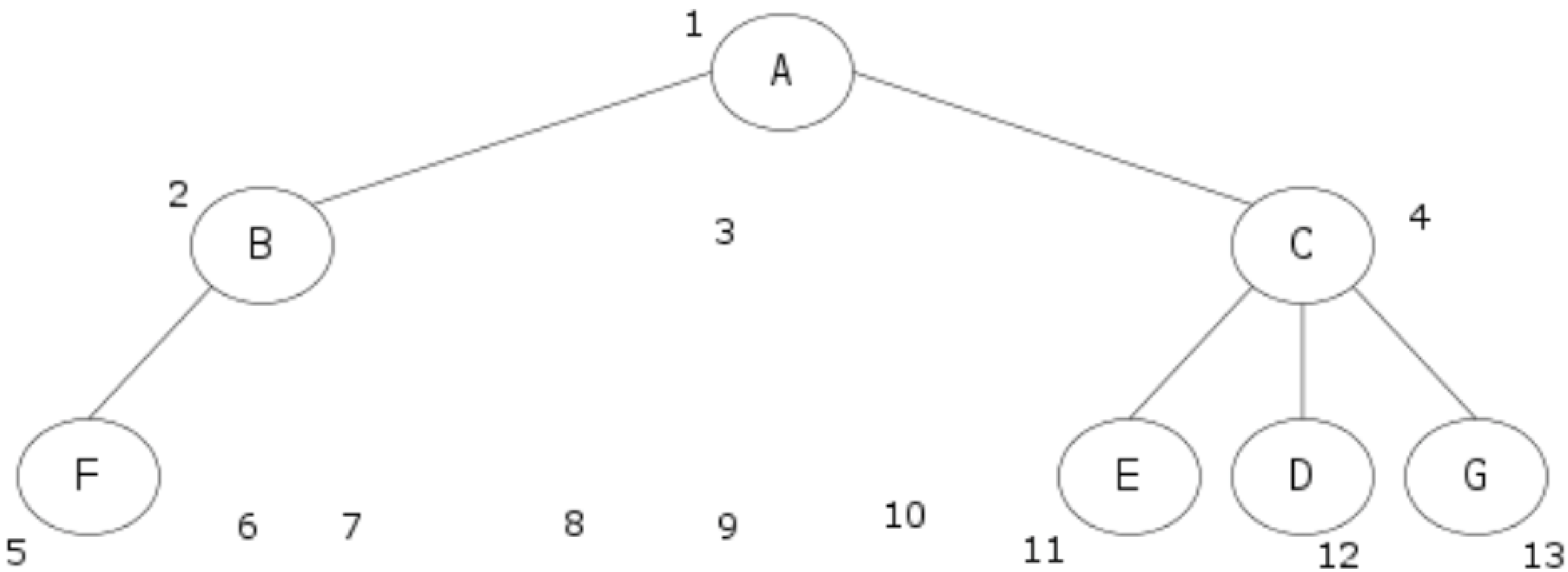
- 2-ary Trees, known as **Binary Trees**
- 3-ary Trees, known as **Ternary Trees**
- 1-ary Trees, known as **Lists**

k-ary Tree Representation

```
class Node {
        private Object data;
        private Node[] children;
        .
        .
        .
}
```



Array Representation



1	A
2	B
3	
4	C
5	F
6	
7	
8	
9	
10	
11	E
12	D
13	G

In a binary tree

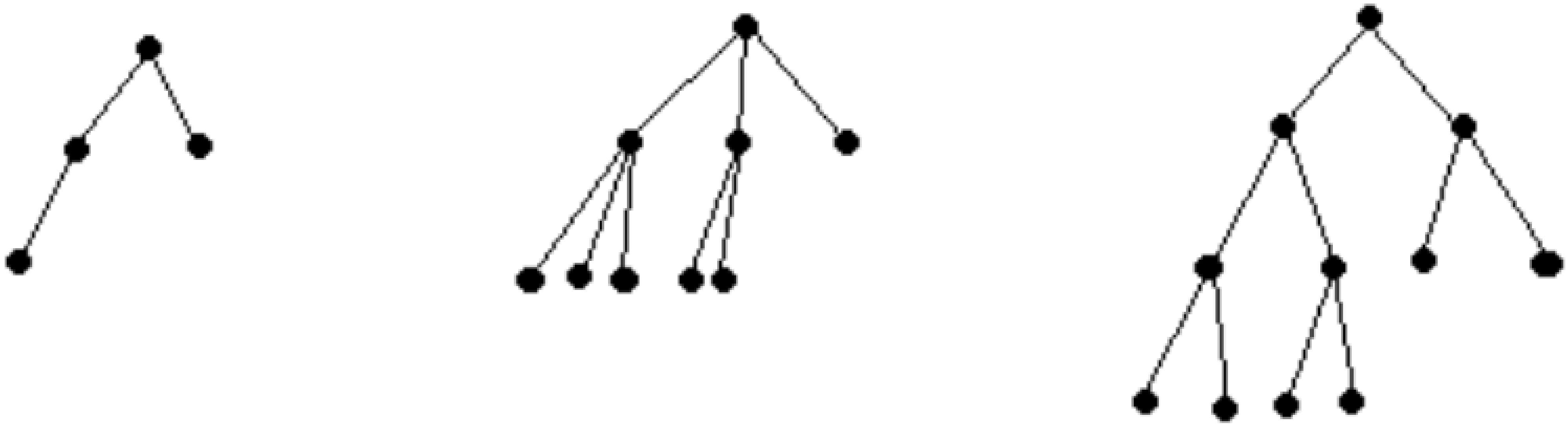
- parent(i) is i / 2
- leftChild(i) is 2i
- rightChild(i) is 2i + 1

In a k-ary tree

- parent(i) is (k + i – 2) / k
- jth child of i is ki – (k–2) + j

Complete k-ary Trees

- Filled out on every level, expect perhaps on the last one
- All nodes on the last level, should be as far to left as possible



Complete trees can be packed into an array with no wasted space

Binary Tree

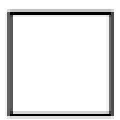
Binary Tree

- A **binary tree** T of n nodes, $n \geq 0$,
  - – either is empty, if $n = 0$

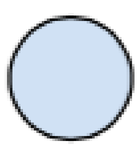
– or consists of a **root node** u and two binary trees u(1) and u(2) of n_1 and n_2 nodes, respectively, such that $n = n + n_1 + n_2$

- We say that u(1) is the **first or left subtree** of T, and u(2) is the **second or right subtree** of T

Binary Tree

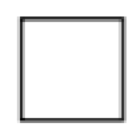


External nodes - have no subtrees (referred to as leaf nodes)



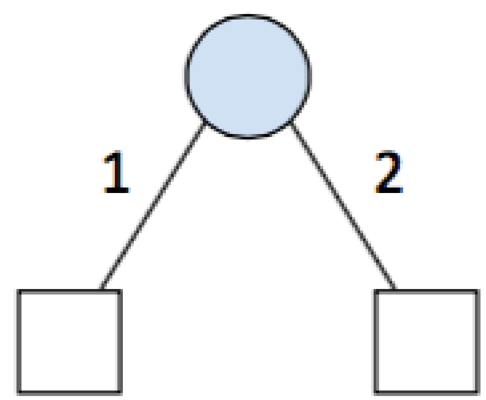
Internal nodes - always have two subtrees

Binary Tree



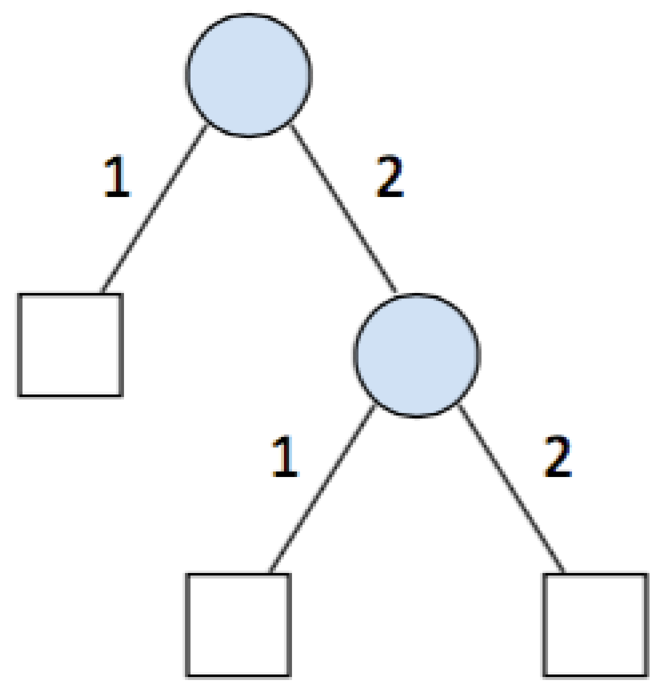
Binary Tree of zero nodes

Binary Tree



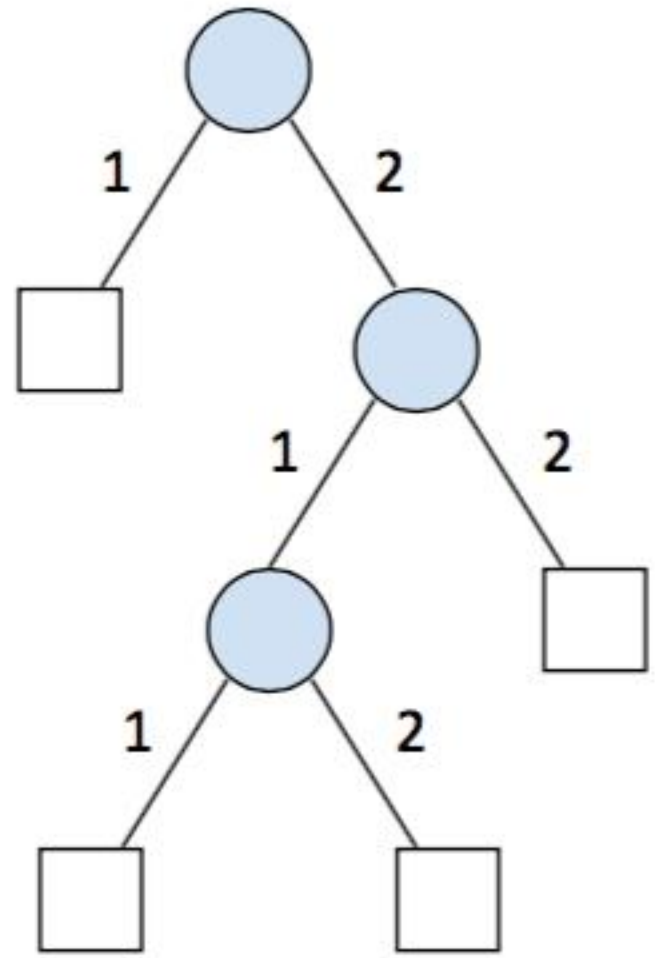
Binary Tree of 1 nodes

Binary Tree



Binary Tree of 2 nodes

Binary Tree



Binary Tree of 3 nodes

Why Binary Trees

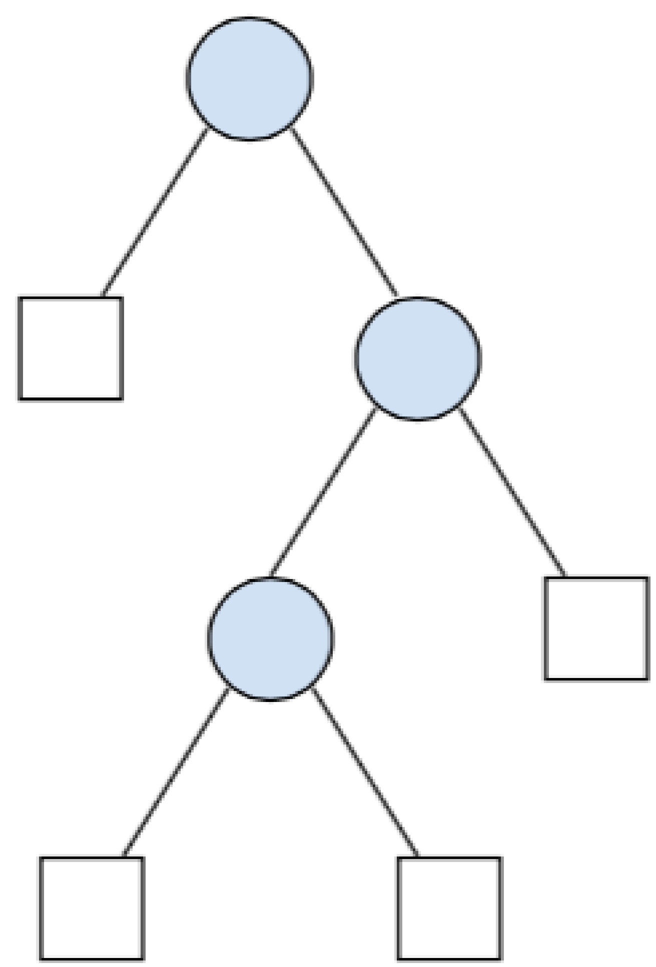
- Binary trees are a bit simpler and easier to understand than trees with a large or unbounded number of children

- Binary tree nodes have an elegant linked representation with "left" and "right" subtrees

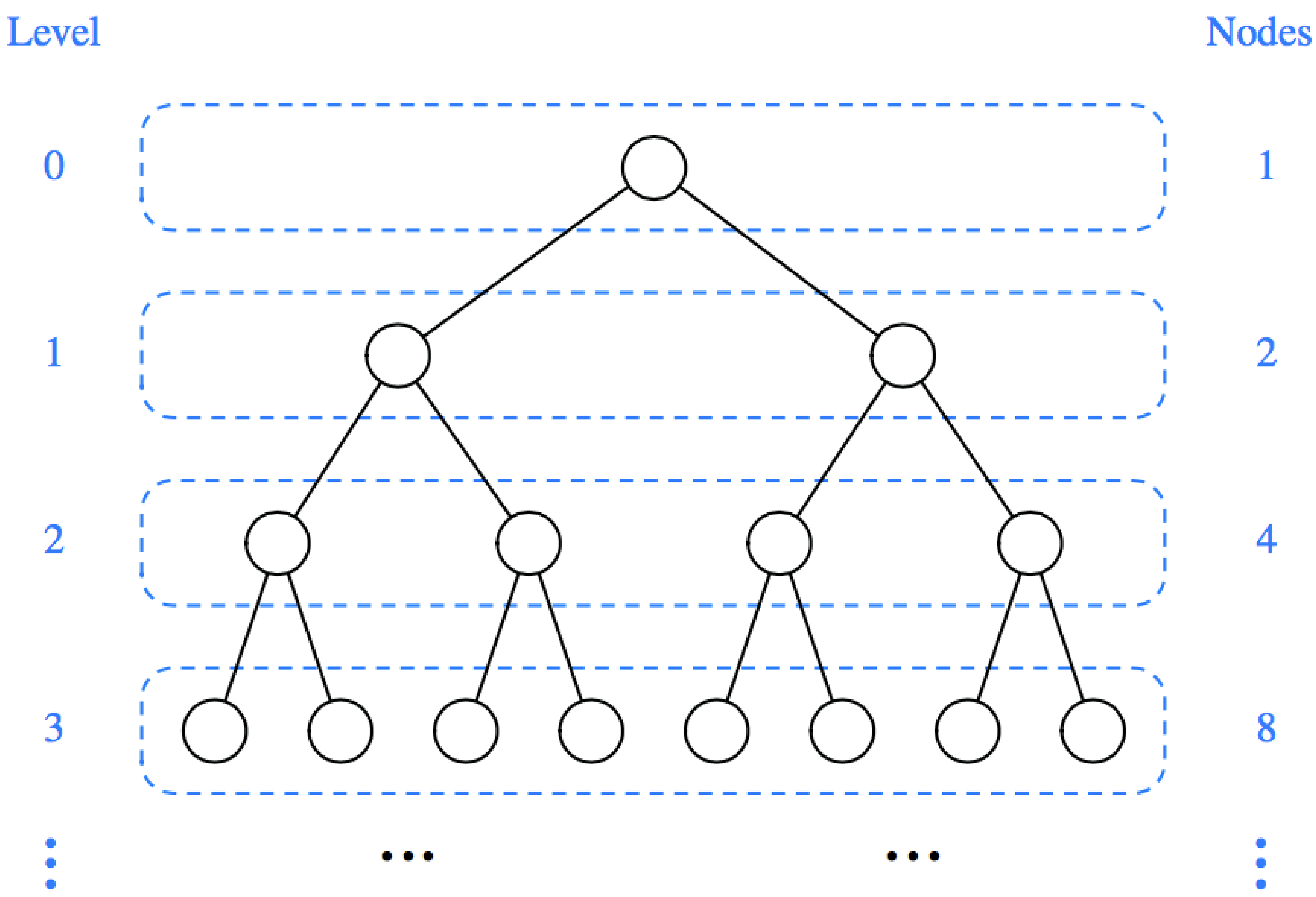
- Binary trees form the basis for efficient representations of sets, dictionaries, and priority queues

Binary Tree Properties

- Let T be a binary tree of size n, n≥0,
- Then, the number of external nodes of T is n+1



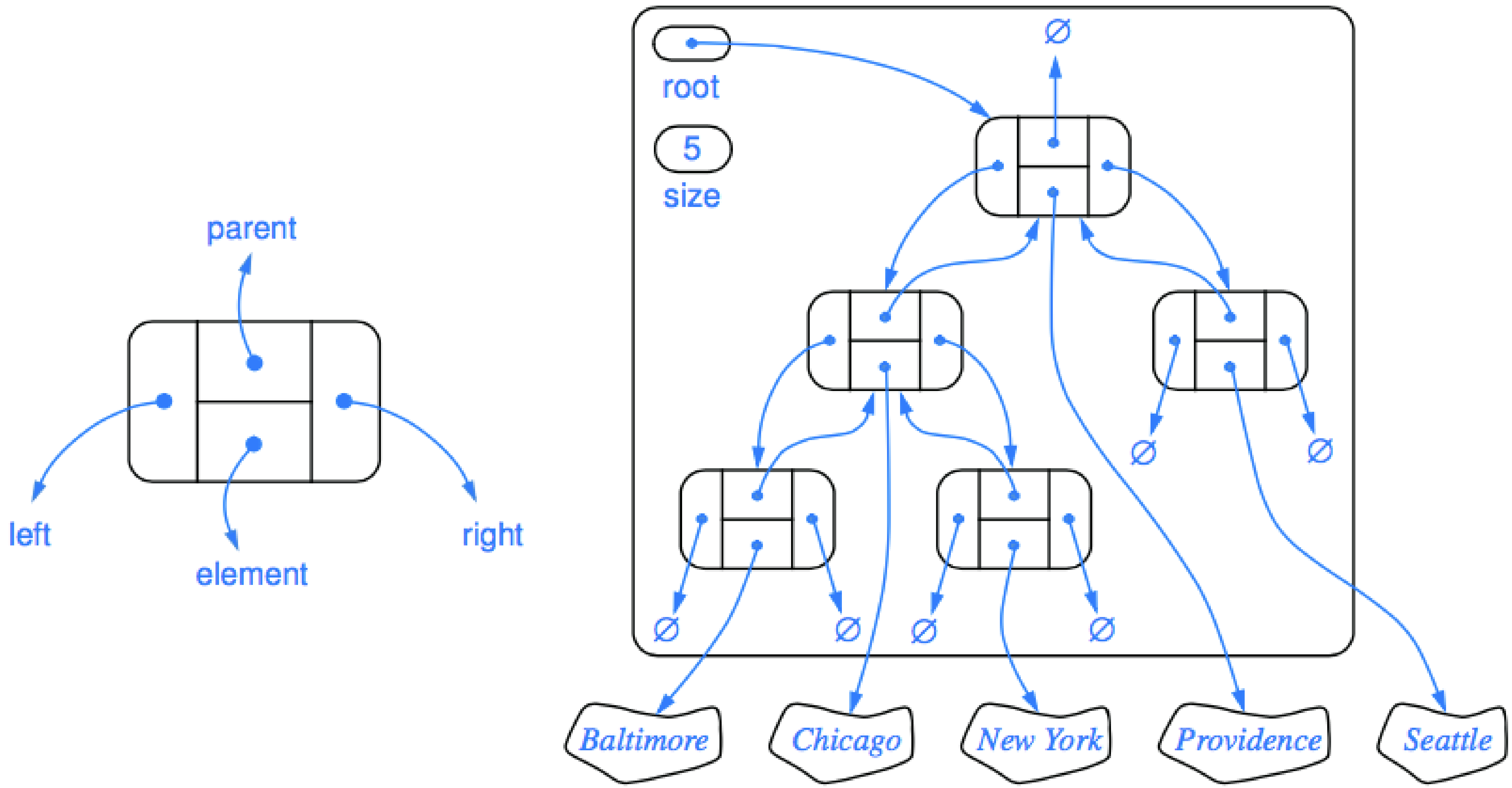
Binary Tree Properties



Maximum number of nodes in the levels of a binary tree

Binary Tree Representations

- Linked Structure



(a) (b)

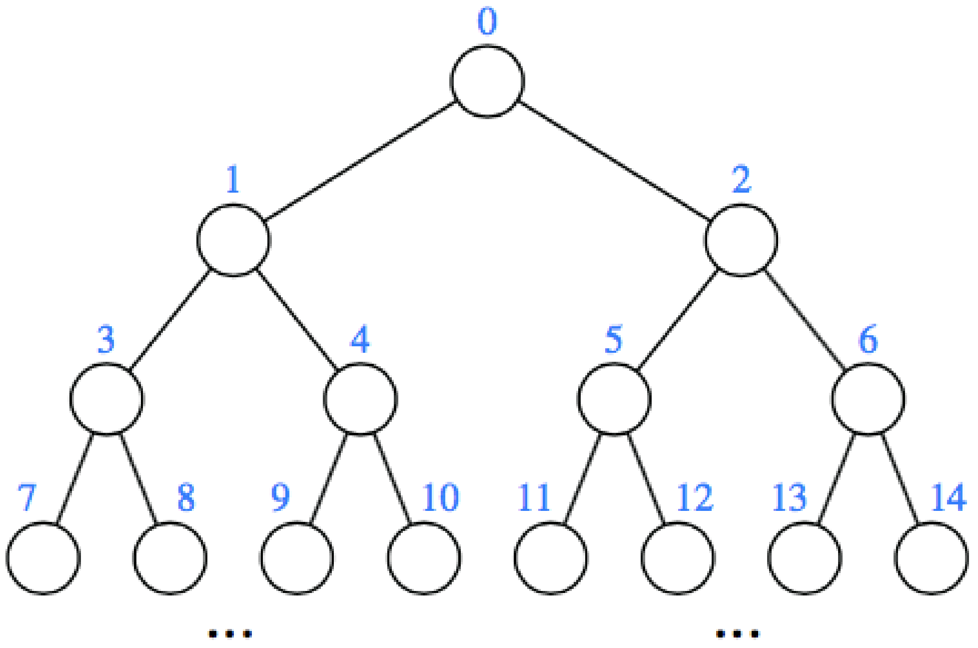
Binary Tree Representations

- Array Based Structure
  - Requires a mechanism for numbering the positions of T
  - For every position p of T , let f (p) be the integer defined as follows
    - If p is the root of T, then f(p) = 0
    - If p is the left child of position q, then f(p) = 2f(q)+1.
    - If p is the right child of position q, then f(p) = 2f(q)+2.

Binary Tree Representations

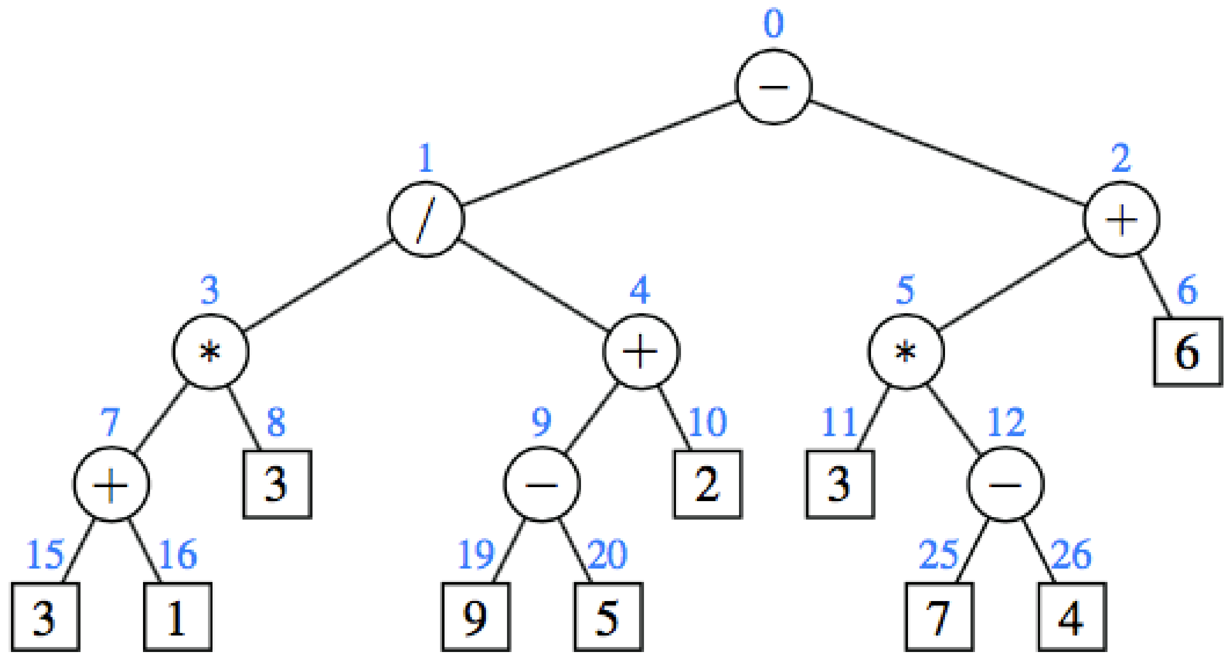
- Array Based Structure

(a)



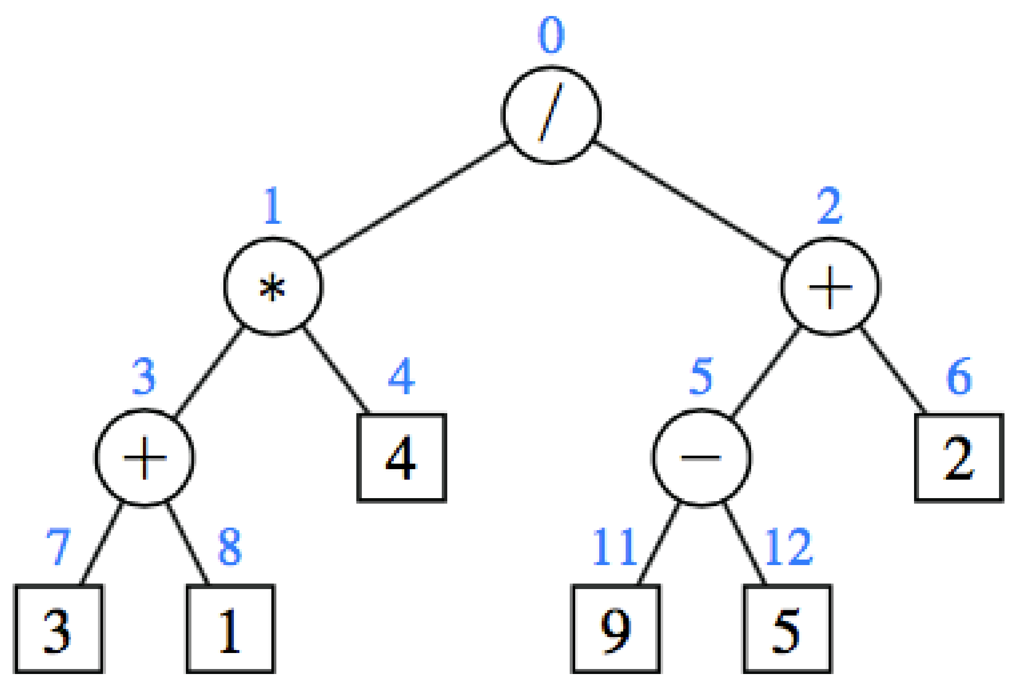
(a) general scheme; (b) an example.

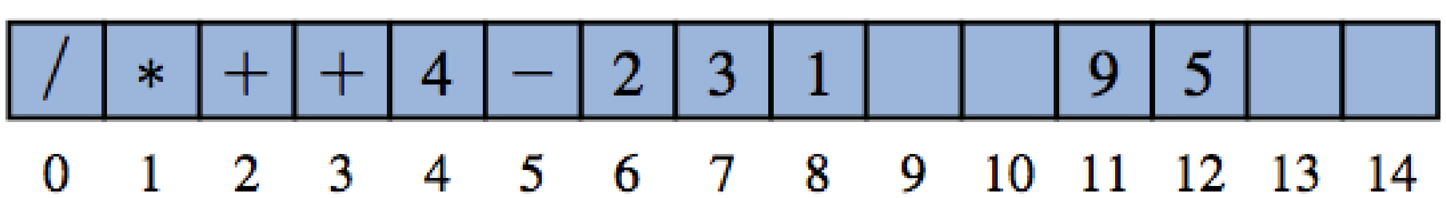
(b)



Binary Tree Representations

- Array Based Structure



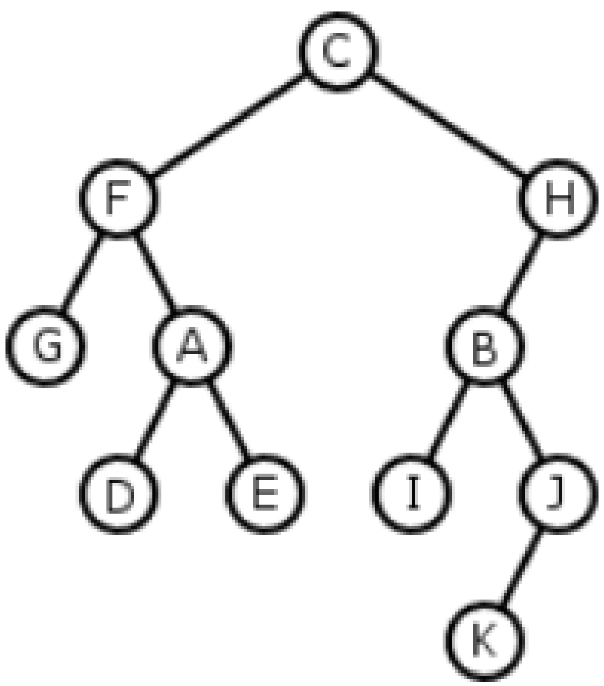


Tree Traversal Algorithms

Tree Traversal Algorithms

- A way of **accessing** or **visiting** all the nodes of T
  - Preorder Traversal –
    - Visit the node, Preorder Left, Preorder Right
  - Postorder Traversal
    - Postorder Left, Postorder Right, Visit the node
  - Inorder Traversal
    - Inorder Left, Visit the node, Inorder Right

Tree Traversals



- **Preorder:** C F G A D E H B I J K
- **Inorder:** G F D A E C I B K J H
- **Postorder:** G D E A F I K J B H C

Preorder Traversal

```
Algorithm preorder(p):
  perform the "visit" action for position p    { this happens before any recursion }
  for each child c in children(p) do
    preorder(c)                                { recursively traverse the subtree rooted at c }
```

Code Fragment 8.12: Algorithm preorder for performing the preorder traversal of a subtree rooted at position p of a tree.

Postorder Traversal

Algorithm postorder(p):
  for each child c in children(p) **do**
    postorder(c)   { recursively traverse the subtree rooted at c }
  perform the "visit" action for position p { this happens after any recursion }

Code Fragment 8.13: Algorithm postorder for performing the postorder traversal of a subtree rooted at position p of a tree.

Inorder Traversal

Algorithm inorder(p):
  if p has a left child lc **then**
    inorder(lc) { recursively traverse the left subtree of p }
  perform the "visit" action for position p
  if p has a right child rc **then**
    inorder(rc) { recursively traverse the right subtree of p }

Code Fragment 8.15: Algorithm inorder for performing an inorder traversal of a subtree rooted at position p of a binary tree.

Implementation

Linked Binary Tree Node

```java
public class LinkedBinaryTreeNode<E>{

    protected E data;
    protected LinkedBinaryTreeNode<E> parent;
    protected LinkedBinaryTreeNode<E> left;
    protected LinkedBinaryTreeNode<E> right;

    /**
     * Constructs a node with a given element
     */
    public LinkedBinaryTreeNode(E data) {
        this.data = data;
        this.parent = null;
        this.left = null;
        this.right = null;
    }

```

Linked Binary Tree Node

```
    /**
     * Constructs a node with a given element and neighbors
     */
    public LinkedBinaryTreeNode(E data, LinkedBinaryTreeNode<E> parent,
            LinkedBinaryTreeNode<E> leftChild, LinkedBinaryTreeNode<E> rightChild) {
        this.data = data;
        this.parent = parent;
        this.left = leftChild;
        this.right = rightChild;
    }

    /**
     * Returns the data stored in this node.
     */
    public E getData() {
        return data;
    }

```

Linked Binary Tree Node

```java
    /**
     * Modifies the data stored in this node.
     */
    public void setData(E data) {
        this.data = data;
    }

    /**
     * Returns the parent of this node, or null if this node is a root.
     */
    public LinkedBinaryTreeNode<E> getParent() {
      return parent;
    }

    /**
     * Sets the parent of this node.
     */
    public void setParent(LinkedBinaryTreeNode<E> parent) {
      this.parent = parent;
    }

```

Linked Binary Tree Node

```java
/**
 * Returns the left child of this node, or null if it does
 * not have one.
 */
public LinkedBinaryTreeNode<E> getLeft() {
  return left;
}
```

Linked Binary Tree Node

```
/**
 * Inserts child as the
 * left child of this node.  If this node already has a left
 * child it is removed.
 * @exception IllegalArgumentException if the child is
 * an ancestor of this node, since that would make
 * a cycle in the tree.
 */
public void setLeft(LinkedBinaryTreeNode<E> childNode) {

// Ensure the child is not an ancestor.
    for (LinkedBinaryTreeNode<E> n = this; n != null; n = n.parent) {
        if (n == childNode) {
            throw new IllegalArgumentException();
        }
    }

    // Break old links, then reconnect properly.
    if (this.left != null) {
        left.parent = null;
    }
    if (childNode != null) {
        childNode.parent = this;
    }
    this.left = childNode;
}
```

Linked Binary Tree Node

```java
/**
 * Returns the right child of this node, or null if it does
 * not have one.
 */
public LinkedBinaryTreeNode<E> getRight() {
  return right;
}
```

Linked Binary Tree Node

```java
    /**
     * Inserts it as the
     * right child of this node.  If this node already has a right
     * child it is removed.
     * @exception IllegalArgumentException if the child is
     * an ancestor of this node, since that would make
     * a cycle in the tree.
     */
    public void setRight(LinkedBinaryTreeNode<E> childNode) {
        // Ensure the child is not an ancestor.
        for (LinkedBinaryTreeNode<E> n = this; n != null; n = n.parent) {
            if (n == childNode) {
                throw new IllegalArgumentException();
            }
        }

        // Break old links, then reconnect properly.
        if (right != null) {
            right.parent = null;
        }
        if (childNode != null) {
            childNode.parent = this;
        }
        this.right = childNode;
    }
```

Linked Binary Tree Node

```java
    /**
     * Removes this node, and all its descendants, from whatever
     * tree it is in.  Does nothing if this node is a root.
     */
    public void removeFromParent() {
        if (parent != null) {
            if (parent.left == this) {
                parent.left = null;
            } else if (parent.right == this) {
                parent.right = null;
            }
            this.parent = null;
        }
    }

    /**
     * Returns the number of children of this node
     *
     */
    public int numChildren(){
    int count = 0;
    if (this.getLeft() != null)
        count++;
    if (this.getRight() != null)
        count++;
    return count;
    }
}
```

Visitor Interface

```java

/**
	* Simple visitor interface.
	* visit can mean anything
	*/
public interface Visitor {

	<E> void visit(LinkedBinaryTreeNode<E> Node);
}

```

Linked Binary Tree

```java

public class LinkedBinaryTree<E> implements Visitor{

    /**
     * Root of the tree.
     */
    protected LinkedBinaryTreeNode<E> root = null;

    /**
     * Number of nodes in the tree.
     */
    private int size = 0;

    /**
     * constructs an empty binary tree.
     */
    public LinkedBinaryTree(){ }
```

Linked Binary Tree

```java

    /**
     * returns the number of nodes in the tree.
     */

    public int size(){
    return size;
    }

    /**
     * Returns the root position of the tree.
     */

    public LinkedBinaryTreeNode<E> root(){
    return root;
    }

    /**
     * Returns whether the tree is empty or not.
     */

    public boolean isEmpty(){
    return size() == 0;
    }
```

Linked Binary Tree

```java
    /**
     * Creates a new node storing element e.
     */

    protected LinkedBinaryTreeNode<E> createNode(E e,
            LinkedBinaryTreeNode<E> parent, LinkedBinaryTreeNode<E> left,
            LinkedBinaryTreeNode<E> right ){

    return new LinkedBinaryTreeNode<E>(e, parent, left, right);
    }

```

Linked Binary Tree

```
/**
 * Place element e at the root of an empty tree.
 */

public LinkedBinaryTreeNode<E> addRoot(E e) throws IllegalStateException {
if(!isEmpty()) throw new IllegalStateException("Tree is not empty");
root = createNode(e, null, null, null);
size = 1;
return root;
}
```

Linked Binary Tree

```java

/**
 * Create a new left child of the node n
 */
public LinkedBinaryTreeNode<E> addLeft(LinkedBinaryTreeNode<E> n, E e){

LinkedBinaryTreeNode<E> child = createNode(e, null, null, null);
n.setLeft(child);
size++;
return child;
}

```

Linked Binary Tree

```java
/**
 * Create a new right child of the node n
 */
public LinkedBinaryTreeNode<E> addRight(LinkedBinaryTreeNode<E> n, E e){

LinkedBinaryTreeNode<E> child = createNode(e, null, null, null);
n.setRight(child);
size++;
return child;
}
```

Linked Binary Tree

```
/**
 * Replaces the element at node n with e
 */
public void set(LinkedBinaryTreeNode<E> n, E e){
n.setData(e);
}
```

Linked Binary Tree

```java
/**
 * Removes the node n and replaces it with its child if any
 */
public void remove(LinkedBinaryTreeNode<E> n) throws IllegalArgumentException{
if (n.numChildren() == 2) throw new IllegalArgumentException("node has two children");
LinkedBinaryTreeNode<E> child = (n.getLeft() != null ? n.getLeft() : n.getRight());
if (child != null)
    child.setParent(n.getParent());
if (n == root)
    root = child;
else{
    LinkedBinaryTreeNode<E> parent = n.getParent();
    if (n == parent.getLeft())
        parent.setLeft(child);
    else
        parent.setRight(child);
}
}
```

Linked Binary Tree

```java
/**
 * Visits the nodes in this tree in preorder.
 */
public void traversePreorder() throws IllegalStateException {
   if (isEmpty()) throw new IllegalStateException("Tree is empty");
   traversePreorder(root);
}

private void traversePreorder(LinkedBinaryTreeNode<E> node){
visit(node);
if (node.getLeft() != null) traversePreorder(node.getLeft());
if (node.getRight() != null) traversePreorder(node.getRight());
}
```

Linked Binary Tree

```
/**
 * Visits the nodes in this tree in postorder.
 */
public void traversePostorder() throws IllegalStateException {
   if (isEmpty()) throw new IllegalStateException("Tree is empty");
   traversePostorder(root);
}

private void traversePostorder(LinkedBinaryTreeNode<E> node){
if (node.getLeft() != null) traversePostorder(node.getLeft());
if (node.getRight() != null) traversePostorder(node.getRight());
visit(node);
}
```

Linked Binary Tree

```
    /**
     * Visits the nodes in this tree in inorder.
     */
    public void traverseInorder() throws IllegalStateException {
       if (isEmpty()) throw new IllegalStateException("Tree is empty");
       traverseInorder(root);
    }

    private void traverseInorder(LinkedBinaryTreeNode<E> node){
    if (node.getLeft() != null) traverseInorder(node.getLeft());
    visit(node);
    if (node.getRight() != null) traverseInorder(node.getRight());
    }
```

Linked Binary Tree

```java
    /**
     * visiting a node in this case means printing its value.
     */
    @Override
    public <E> void visit(LinkedBinaryTreeNode<E> Node) {
        // TODO Auto-generated method stub
        System.out.print(""+Node.getData()+" ");

    }
}

```

Tree Driver Class

```java

/**
     * Simple driver class for testing our binary tree implementation.
     */

public class TreeDriver {

    /**
    * Makes an application which adds three string elements to a binary tree.
    * and performs tree traversals
    */
    public static void main(String args[]){

        LinkedBinaryTree<String> tree = new LinkedBinaryTree<String>();
        LinkedBinaryTreeNode<String> node;

```

Tree Driver Class

```java
        //adding the root
        node = tree.addRoot("-");
        //adding the left child
        tree.addLeft(node, "/");
        //adding the right child
        tree.addRight(node, "+");

        //preorder traversal
        tree.traversePreorder();
        System.out.println();

        //postorder traversal
        tree.traversePostorder();
        System.out.println();

        //inorder traversal
        tree.traverseInorder();


    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> TreeDriver [Java Application] /System/Library/Java/JavaVirtu
- / +
/ + -
/ - +
```

- The intuition behind this implementation was to show the working of a linked binary tree in a simplified manner

- For a more through implementation, please consult Chapter. 8 of your textbook

Observations

- Did you notice that there was no generic add method in our implementation?

- Why do you think it is so?