Trees

Readings: Chapter 5 and Chapter 8

CSE214: Computer Science II
Fall 2015

CHEN-WEI WANG
Important Background Reading

• You will learn and be required to write algorithms that are written using recursion.

• Chapter 5 discusses recursion, and you are required to study this chapter on your own.

• By next Friday’s (November 13) Quiz, you are expected to know about recursion.
General Trees

• A *linear* data structure is a sequence, where stored objects can be related via the “*before*” and “*after*” relationships. e.g., arrays, singly-linked lists, and doubly-linked lists

• A *tree* is a *non-linear* collection of nodes/positions.
  ○ Each node stores some data object.
  ○ Nodes stored in a *tree* is organized in a *non-linear* manner.
  ○ In a *tree*, the relationships between stored objects are *hierarchical*: some objects are “*above*” others, and some are “*below*” others.

• The main terminology for the *tree* data structure comes from that of family trees: parents, siblings, children, ancestors, descendants.
General Trees: Terminology (1)

- **root of tree**: top element of the tree
e.g., *root* of the above family tree: David

- **parent of node** \( v \): node immediately above and connected to \( v \)
e.g., *parent* of Vanessa: Elsa

- **children of node** \( v \): nodes immediately below and connected to \( v \)
e.g., *children* of Elsa: Shirley, Vanessa, and Peter
  
e.g., *children* of Ernesto: \( \emptyset \)
General Trees: Terminology (2)

- **ancestors of node** $v$: $v$ + $v$’s parent + $v$’s grand parent + ...  
  e.g., **ancestors** of Vanessa: Vanessa, Elsa, Chris, and David  
  e.g., **ancestors** of David: David

- **descendants of node** $v$: $v$ + $v$’s children + $v$’s grand children + ...  
  e.g., **descendants** of Vanessa: Vanessa  
  e.g., **descendants** of David: the entire family tree
siblings of node v: nodes whose parents are the same as v’s
  e.g., siblings of Vanessa: Shirley and Peter
subtree rooted at v: a tree formed by all descendant of v
external nodes (leaves): nodes that have no children
  e.g., leaves of the above tree: Ernesto, Anna, Shirley, Vanessa, Peter
internal nodes: nodes that has at least one children
  e.g., non-leaves of the above tree: David, Chris, Elsa
General Trees: Terminology (4)

- **an edge of tree**: a *pair* of parent and child nodes
  e.g., (David, Chris), (Chris, Elsa), (Elsa, Peter) are three edges

- **a path of tree**: a *sequence* of nodes where any two consecutive
  nodes form an *edge*
  e.g., ⟨David, Chris, Elsa, Peter⟩ is a path
○ **depth of a node** $v$ (distance from root): number of *edges* from the root to $v$; alternatively, number of *ancestors* of $v$, except itself
  
  e.g., *depth* of David (root): 0
  
  e.g., *depth* of Shirley, Vanessa, and Peter: 3

○ **height of tree**: maximum *depth* of its nodes
  
  e.g., Shirley, Vanessa, and Peter have the maximum depth
General Trees: Example Node Depths

Balanced Binary Tree
- every node at depths 0, 1, ..., \( d_{\text{max}} - 2 \) has two children; nodes at depth \( (d_{\text{max}} - 1) \) may have two, one, or no children; nodes at depth \( h \) have no children.

A balanced binary tree

An ill-balanced binary tree

\( d=0 \)

A

\( d=1 \)

B

C

\( d=2 \)

D

E

F

G

\( d=3 \)

H

I

J

\( d=4 \)

G

H

I

J
A **tree** $T$ is a set of **nodes** which satisfy the following **parent-child** properties:

1. If tree $T$ is **empty**, then it does not contain nodes.
2. If tree $T$ is **nonempty**, then it contains **at least** a special node that has no parent: the **root of $T$**.
3. Each node $v$ of $T$ that is different from the root has a **unique parent node** $w$.
4. Every node with parent $w$ is a **child** of $w$. 
General Tree: Important Characteristics

There is a *single unique path* along the edges from the *root* to any particular node.

![Diagram of legal tree organization](image1)

*legal tree organization*

![Diagram of illegal tree organization (nontrees)](image2)

*illegal tree organization (nontrees)*
A tree is **ordered** if there is a meaningful *linear* order among the *children* of each node.
General Trees: Unordered Trees

A tree is **unordered** if the order among the *children* of each node does not matter.
Tree Operation (1.1):
Computing the Depth of a Node

Given a position (or a node) $p$, its depth is defined as:
- If $p$ is the root, then $p$’s depth is 0.
- Otherwise, $p$’s depth is the depth of $p$’s parent plus one.

```java
public int depth(Position<E> p) {
    if (isRoot(p)) { return 0; }
    else {
        return 1 + depth(parent(p));
    }
}
```
Tree Operation (1.2): Computing the Depth of a Node

Consider the family tree example:

\[
\text{depth}(Vanessa) \\
= \{ \text{parent}(Vanessa) \text{ is Elsa} \} \\
= 1 + \text{depth}(Elsa) \\
= \{ \text{parent}(Elsa) \text{ is Chris} \} \\
= 1 + 1 + \text{depth}(Chris) \\
= \{ \text{parent}(Chris) \text{ is David} \} \\
= 1 + 1 + 1 + \text{depth}(David) \\
= \{ \text{David is the root} \} \\
= 1 + 1 + 1 + 0 \\
= 3
\]
Tree Operation (2.1):
Computing the Height of Tree

Given a position (or a node) $p$, the height of the subtree rooted at $p$ is defined as:
- If $p$ is a leaf, then the height of subtree rooted at $p$ is 0.
- Otherwise, the height of subtree rooted as $p$ is the maximum height of subtrees rooted at $p$’s children plus one.

```java
public int height(Position<E> p) {
    if(isExternal(p)) { return 0; }
    else {
        int h = 0;
        for(Position<E> c: children(p)) {
            h = Math.max(h, 1 + height(c));
        }
        return h;
    }
}
```
Tree Operation (2.2): Computing the Height of Tree

Consider the family tree example:

\[ \text{height}(\text{subtree rooted at Chris}) \]
\[ = \{ \text{Chris is not a leaf} \} \]
\[ \text{MAX} \left( \begin{array}{c}
1 + \text{height}(\text{subtree rooted at Elsa}), \\
1 + \text{height}(\text{subtree rooted at Anna})
\end{array} \right) \]
\[ = \{ \text{Elsa is not a leaf, Anna is a leaf} \} \]
\[ \text{MAX} \left( \begin{array}{c}
1 + \text{MAX} \left( \begin{array}{c}
1 + \text{height}(\text{subtree rooted at Shirley}), \\
1 + \text{height}(\text{subtree rooted at Vanessa}), \\
1 + \text{height}(\text{subtree rooted at Peter})
\end{array} \right), \\
1 + 0
\end{array} \right) \]
\[ = \{ \text{Shirley, Vanessa, and Peter are all leaves} \} \]
\[ \text{MAX} \left( \begin{array}{c}
1 + \text{MAX} \left( \begin{array}{c}
1 + 0, \\
1 + 0, \\
1 + 0
\end{array} \right), \\
1 + 0
\end{array} \right) \]
\[ = 2 \]
Exercises (1)

- Implement following recursive algorithm:
  \[
  \text{ArrayList<}E\text{> ancestors(Position<}E\text{> } p)
  \]
  which returns the list of ancestors of a given position \( p \).

- Implement following recursive algorithm:
  \[
  \text{ArrayList<}E\text{> descendants(Position<}E\text{> } p)
  \]
  which returns the list of descendants of a given position \( p \).
Binary Trees

- A **binary tree** is an **ordered** tree which satisfy the following properties:
  1. Each node has **at most two** children.
  2. Each child node is labeled as either a **left child** or a **right child**.
  3. A **left child** precedes a **right child** in the order of children of a node.
Binary Trees: Terminology (1)

For an *internal* node:

- Subtree rooted at its *left child*, if any, is called *left subtree*.
- Subtree rooted at its *right child*, if any, is called *right subtree*.

E.g.,

Node A has a *left subtree* rooted at node B and a *right subtree* rooted at node C.
A *binary* tree is either:

- An *empty* tree; or
- A *nonempty* tree with a root node $r$ that
  - has a *binary subtree* rooted at its left child, if any
  - has a *binary subtree* rooted at its right child, if any
In a binary tree, the set of nodes with the same depth $d$ are said to be at level $d$. 
Background (1): Sum of Geometric Sequence

\[
\sum_{k=0}^{n-1} (ar^k) = a + ar + ar^2 + \cdots + ar^{n-1} = a \left( \frac{r^n - 1}{r - 1} \right)
\]

e.g.,

\[
1 + 2 + 4 + 8 + 16 = 1 \left( \frac{2^5 - 1}{2 - 1} \right) = 2^5 - 1 = 31
\]
Binary Trees: Properties (1)

Given a binary tree with *height* $h$:

- Maximum number of nodes at *level 0*: $2^0 = 1$
- Maximum number of nodes at *level 1*: $2^1 = 2$
- Maximum number of nodes at *level 2*: $2^2 = 4$
  
  \[ \ldots \]
- Maximum number of nodes at *level h*: $2^h$

- Maximum *total* number of nodes:
  \[ 2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1 \]
Given a binary tree with \textit{height} \( h \), the \textit{number of nodes} \( n \) is bounded as:

\[
    h + 1 \leq n \leq 2^{h+1} - 1
\]

- Shape of BT with \textit{minimum} # of nodes?
  A “one-path” tree (each internal nodes has exactly one child)
- Shape of BT with \textit{maximum} # of nodes?
  A tree completely filled at each level
Given a binary tree with \( n \) modes, the \textit{height} \( h \) is bounded as:

\[
\log(n + 1) - 1 \leq h \leq n - 1
\]

- **Shape of BT with \textit{minimum} height?**
  A tree completely filled at each level

\[
\begin{align*}
n & \quad = \quad 2^{h+1} - 1 \\
\Leftrightarrow n + 1 & \quad = \quad 2^{h+1} \\
\Leftrightarrow \log(n + 1) & \quad = \quad h + 1 \\
\Leftrightarrow \log(n + 1) - 1 & \quad = \quad h
\end{align*}
\]

- **Shape of BT with \textit{maximum} height?**
  A “one-path” tree (each internal nodes has exactly one child)
Given a binary tree with height $h$, the number of external nodes $n_E$ is bounded as:

$$1 \leq n_E \leq 2^h$$

- Shape of BT with $minimum$ # of external nodes?
  A tree with the root node only
- Shape of BT with $maximum$ # of external nodes?
  A tree whose bottom level (with depth $h$) is completely filled
Given a binary tree with height $h$, the number of internal nodes $n_I$ is bounded as:

$$h \leq n_I \leq 2^h - 1$$

- Shape of BT with **minimum** # of internal nodes?
  A “one-path” tree (each internal nodes has exactly one child)
- Shape of BT with **maximum** # of internal nodes?
  A tree whose $\leq h - 1$ levels are all completely filled
A binary tree is considered as proper if all its internal node has two children.
Given a nonempty proper binary tree with the number of internal nodes $n_I$ and the number of external nodes $n_E$:

$$n_E = n_I + 1$$

Proof by mathematical induction:

- **Base Case:** When proper BT contains only the root (being an external node), $n_E = 1$ and $n_I = 0$.

- **Inductive Case:**
  - Assume that a proper BT with $n$ nodes ($n > 1$) has $n_I$ internal nodes and $n_E$ external nodes, such that $n_E = n_I + 1$.
  - The only way to create a larger proper BT (with $n'_E$ and $n'_I$) is to:
    1. Convert an external node into an internal node.
    2. Add two external child nodes to the new internal node.

$$n'_E = (n_I + 1) - 1 + 2$$ and $$n'_I = n_I + 1$$
A binary tree with height $h$ is considered as complete if:

- Nodes with depth $\leq h - 2$ has two children.
- Nodes with depth $h - 1$ may have zero, one, or two children.
- Children of nodes with depth $h - 1$ are filled from left to right.

**Q1:** Minimum # of nodes of a complete BT? \[ (2^h - 1) + 1 = 2^h \]

**Q2:** Maximum # of nodes of a complete BT? \[ 2^{h+1} - 1 \]
Binary Trees: Terminology (5)

A binary tree with height $h$ (i.e., maximum depth of nodes) is considered as **full** if:

- All nodes with depth $\leq h - 1$ has two children
- All *leaves* are with the same *depth* $h$

**Q1:** *Minimum* # of nodes of a complete BT? $2^{h+1} - 1$

**Q2:** *Maximum* # of nodes of a complete BT? $2^{h+1} - 1$
Binary Trees: Application (1)

A decision tree is a binary tree used to express the decision-making process:

- Each internal node has two children (yes and no).
- Each external node represents a decision.

Are you nervous?

- Yes: Savings account.
- No: Will you need to access most of the money within the next 5 years?
  - Yes: Money market fund.
  - No: Are you willing to accept risks in exchange for higher expected returns?
    - Yes: Stock portfolio.
    - No: Diversified portfolio with stocks, bonds, and short-term instruments.
An arithmetic expression can be represented using a binary tree:

- Each internal node denotes an operator (unary or binary).
- Each external node denotes an operand (i.e., a number).

To evaluate the expression that is represented by a binary tree, certain traversal over the entire is required.
Tree Traversal Algorithms: Definition

- **Traversal** of a tree $T$ is a systematic way of visiting all the nodes/positions of $T$.
- The visit of each node/position may be associated with an action: e.g.,
  - print the node element
  - determine if the node element satisfies certain property
  - accumulate the node element value to some global counter
Tree Traversal Algorithms: Common Types

Three common traversal orders:

- **Preorder**: Visit parent, then visit child subtrees.

  ```
  preorder(p)
  visit and act on position p
  for child c: children(p) { preorder(c) }
  ```

- **Postorder**: Visit child subtrees, then visit parent.

  ```
  postorder(p)
  for child c: children(p) { postorder(c) }
  visit and act on position p
  ```

- **Inorder** (for BT): Visit left subtree, then parent, then right subtree.

  ```
  inorder(p)
  if (p has a left child lc) { inorder(lc) }
  visit and act on position p
  if (p has a right child rc) { inorder(rc) }
  ```
**Preorder**: Visit parent, then visit child subtrees.

\[ \text{preorder}(p) \]

- visit and act on position \( p \)
- for child \( c \): \( \text{children}(p) \) \{ \text{preorder}(c) \}
Tree Traversal Algorithms: Postorder

**Postorder**: Visit child subtrees, then visit parent.

\[
\text{postorder}(p) \\
\text{for child } c: \text{ children}(p) \{ \text{postorder}(c) \} \\
\text{visit and act on position } p
\]
Inorder (for BT): Visit left subtree, then parent, then right subtree.

\[ \text{inorder}(p) \]

- if (\( p \) has a left child \( \text{lc} \)) { \text{inorder}(\text{lc}) \}
- visit and act on position \( p \)
- if (\( p \) has a right child \( \text{rc} \)) { \text{inorder}(\text{rc}) \}
Trees in Java: Class Hierarchy

- Tree
  - AbstractTree
    - AbstractBinaryTree
      - LinkedBinaryTree
  - BinaryTree
Trees in Java: Class Hierarchy

- Tree
  - AbstractTree
  - BinaryTree
    - AbstractBinaryTree
      - LinkedBinaryTree
General Tree: Java Interface

```java
public interface Position<E> { /* a tree node */
    E getElement();
}

public interface Tree<E> extends Iterable<E> { 
    Iterable<Position<E>> positions(); /* iterate through nodes */
    Iterator<E> iterator(); /* iterate through data objects */
    int size();
    boolean isEmpty();
    Position<E> root();
    Position<E> parent(Position<E> p);
    Iterable<Position<E>> children(Position<E> p);
    int numChildren(Position<E> p);
    boolean isInternal(Position<E> p);
    boolean isExternal(Position<E> p);
    boolean isRoot(Position<E> p);
}
```
Trees in Java: Class Hierarchy

- Tree
  - AbstractTree
  - BinaryTree
    - AbstractBinaryTree
      - LinkedBinaryTree
• **Methods** `root`, `parent`, and `children` remain *abstract*.  
  ○ These methods will be implemented by lower sub-classes.
• **Other methods inherited from the** `Tree` **interface are implemented using these** *abstract* **methods.**  
  ○ `AbstractTree` cannot be instantiated directly!  
  ○ A *concrete* class may extend `AbstractTree` and implement these abstract methods.  
  ○ When instantiating this *concrete* class at runtime, calling these methods will work according to *dynamic binding*.  
• **New methods are also defined:** `depth`, `height`, `preorder`, and `postorder`.  

public abstract class AbstractTree<E> implements Tree<E> {
    public Iterable<Position<E>> preorder() {
        List<Position<E>> snapshot = new ArrayList<>();
        if (!isEmpty())
            preorderSubtree(root(), snapshot);
        return snapshot;
    }

    /* recursive helper method */
    private void preorderSubtree(Position<E> p, List<Position<E>> snapshot) {
        snapshot.add(p);
        for (Position<E> c : children(p)) {
            preorderSubtree(c, snapshot);
        }
    }
}

General Tree: Java Abstract Class (2)
public Iterable<Position<E>> postorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if (!isEmpty())
        postorderSubtree(root(), snapshot);
    return snapshot;
}

/*/ recursive helper method */
private void postorderSubtree (Position<E> p, List<Position<E>> snapshot) {
    for (Position<E> c : children(p)) {
        postorderSubtree(c, snapshot);
    }
    snapshot.add(p);
}
public Iterable<Position<E>> positions() {
    return preorder();
}

public int size() {
    int count = 0;
    for (Position p : positions()) count++;
    return count;
}

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

public Iterator<E> iterator() {
    ArrayList<E> elements = new ArrayList<>();
    for (Position<E> p : positions()) {
        elements.add(p.getElement());
    }
    return elements;
}
public int numChildren(Position<E> p) {
    int count=0;
    for (Position child : children(p)) count++;
    return count;
}

public boolean isInternal(Position<E> p) {
    return numChildren(p) > 0;
}

public boolean isExternal(Position<E> p) {
    return numChildren(p) == 0;
}

public boolean isRoot(Position<E> p) {
    return p == root();
}
public int depth(Position<E> p) {
    if (isRoot(p)) {
        return 0;
    }
    else {
        return 1 + depth(parent(p));
    }
}

public int height(Position<E> p) {
    int h = 0;
    for (Position<E> c : children(p)) {
        h = Math.max(h, 1 + height(c));
    }
    return h;
}
Trees in Java: Class Hierarchy

- Tree
- AbstractTree
- BinaryTree
- AbstractBinaryTree
- LinkedBinaryTree
public interface Position<E> { /* a tree node */
    E getElement();
}

public interface Tree<E> extends Iterable<E> { /* general tree */
    ...
}

public interface BinaryTree<E> extends Tree<E> { // Binary tree
    Position<E> left(Position<E> p);
    Position<E> right(Position<E> p);
    Position<E> sibling(Position<E> p);
}
Binary Tree: Java Abstract Class (1)

- Methods `root`, `parent`, and `children` remain `abstract`.
  - These methods will be implemented by lower sub-classes.
- Definitions of other methods inherited from the `BinaryTree` (and directly from `Tree`) interface are inherited from the `AbstractTree` class.
  - `AbstractBinaryTree` still cannot be instantiated directly!
  - A `concrete` class may extend `AbstractBinaryTree` and implement these abstract methods.
  - When instantiating this `concrete` class at runtime, calling these methods will work according to `dynamic binding`.
- New methods are also `defined`: `sibling` and `inorder`.
- Some inherited are `redefined (overridden)`: `numChildren` and `children`
public abstract class AbstractBinaryTree<E> 
    extends AbstractTree<E> 
    implements BinaryTree<E> 

public Position<E> sibling(Position<E> p) {
    Position<E> parent = parent(p);
    /* p must be the root */
    if (parent == null) {
        return null;
    }
    /* p is a left child */
    if (p == left(parent)) { /* why not else if? */
        return right(parent); /* right child might be null */
    }
    /* p is a right child */
    else {
        return left(parent); /* left child might be null */
    }
}
public Iterable<Position<E>> inorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if (!isEmpty()) {
        inorderSubtree(root(), snapshot);
    }
    return snapshot;
}

private void inorderSubtree(Position<E> p, List<Position<E>> snapshot) {
    if (left(p) != null) {
        inorderSubtree(left(p), snapshot);
    }
    snapshot.add(p);
    if (right(p) != null) {
        inorderSubtree(right(p), snapshot);
    }
}

public Iterable<Position<E>> positions() { return inorder(); }
public int numChildren(Position<E> p) {
    int count=0;
    if (left(p) != null)
        count++;
    if (right(p) != null)
        count++;
    return count;
}

public Iterable<Position<E>> children(Position<E> p) {
    List<Position<E>> snapshot = new ArrayList<>(2);
    if (left(p) != null)
        snapshot.add(left(p));
    if (right(p) != null)
        snapshot.add(right(p));
    return snapshot;
}
Binary Tree: Java Abstract Class (5)

• **Exercise:** Override/Redefine the `preorder` method inherited from `AbstractTree`.

• **Exercise:** Override/Redefine the `postorder` method inherited from `AbstractTree`.
Trees in Java: Class Hierarchy

- Tree
  - AbstractTree
  - BinaryTree
    - AbstractBinaryTree
    - LinkedBinaryTree
Linked Binary Tree: Linked Object Structure

- Parent
- Element
- Left
- Right

Root: ∅

Baltimore Chicago New York Providence Seattle

Size: 5
class Node<E> implements Position<E> {
    private E element;
    private Node<E> parent;
    private Node<E> left;
    private Node<E> right;

    public Node(
        E e,
        Node<E> above,
        Node<E> leftChild,
        Node<E> rightChild) {
        element = e;
        parent = above;
        left = leftChild;
        right = rightChild;
    }
}
/* accessor methods */
public E getElement() { return element; }
public Node<E> getParent() { return parent; }
public Node<E> getLeft() { return left; }
public Node<E> getRight() { return right; }

/* mutator methods */
public void setElement(E e) {
    element = e;
}
public void setParent(Node<E> parentNode) {
    parent = parentNode;
}
public void setLeft(Node<E> leftChild) {
    left = leftChild;
}
public void setRight(Node<E> rightChild) {
    right = rightChild;
}
} /* end Node class */
public class LinkedBinaryTree\<E\> extends AbstractBinaryTree\<E\> {

    Node\<E\> root = null;
    private int size = 0;

    public LinkedBinaryTree() {
    }

    public int size() {
        return size;
    }

    public Position\<E\> root() {
        return root;
    }
}
public Position<E> parent(Position<E> p) {
    Node<E> node = (Node<E>) p;
    return node.getParent();
}

public Position<E> left(Position<E> p) {
    Node<E> node = (Node<E>) p;
    return node.getLeft();
}

public Position<E> right(Position<E> p) {
    Node<E> node = (Node<E>) p;
    return node.getRight();
}
public Position<E> addRoot(E e) throws IllegalStateException {
    if (!isEmpty()) {
        throw new IllegalStateException(
            "Tree is not empty");
    }
    root = new Node<E>(e, null, null, null);
    size = 1;
    return root;
}

double set(Position<E> p, E e) {
    Node<E> node = (Node<E>) p;
    E temp = node.getElement();
    node.setElement(e);
    return temp;
}
```java
public Position<E> addLeft(Position<E> p, E e)
    throws IllegalArgumentException {
    Node<E> parent = (Node<E>) p;
    if (parent.getLeft() != null) {
        throw new IllegalArgumentException(
            "p already has a left child");
    }
    Node<E> child = new Node<E>(e, parent, null, null);
    parent.setLeft(child);
    size++;
    return child;
}
```
Exercise: Implement

```java
Position<E> addRight(Position<E> p, E e)
```
public void attach(
    Position<E> p,
    LinkedBinaryTree<E> t1,
    LinkedBinaryTree<E> t2) throws IllegalArgumentException {
    Node<E> node = (Node<E>) p;
    if (isInternal(p))
        throw new IllegalArgumentException("p must be a leaf");
    size += t1.size() + t2.size();
    if (!t1.isEmpty()) {
        t1.root.setParent(node);
        node.setLeft(t1.root);
        t1.root = null;
        t1.size = 0;
    }
    if (!t2.isEmpty()) {
        t2.root.setParent(node);
        node.setRight(t2.root);
        t2.root = null;
        t2.size = 0;
    }
}
public E remove(Position<E> p) throws IllegalArgumentException {
    Node<E> node = (Node<E>) p;
    if (numChildren(p) == 2)
        throw new IllegalArgumentException("p has two children");
    Node<E> child =
        (node.getLeft() != null ? node.getLeft() : node.getRight());
    if (child != null)
        child.setParent(node.getParent());
    if (node == root)
        root = child;
    else {
        Node<E> parent = node.getParent();
        if (node == parent.getLeft()) { parent.setLeft(child); }
        else { parent.setRight(child); }
    }
    size--;
    E temp = node.getElement();
    return temp;
} //---------- end of LinkedBinaryTree ----------
A Note on Recursive Helper Method (1)

Whether to define a **private recursive helper method** or not depends on *how you want the user to use your code*:

A recursive helper method contains *additional parameters* that allow the computation to focus on a particular *sub-range* of input.

- If you think the user may specify the “*sub-range*” of input that is to be computed, then a recursive method is not necessary.
  - e.g., `ArrayList<Position<E>> ancestors(Position<E> p)`
  - e.g., `int depth(Position<E> p)`

- If you think the user should compute on the “*entire range*”, then a recursive that is hidden from the user should be defined.
  - e.g., for `preorder()`, which always starts with the root, we need a helper method `preorderSubtree(Position<E> p, ...)` to handle the intermediate subtrees.
public static boolean isPalindrome(String word) {
    if (word.length() < 2) {
        return true;
    }
    else if (word.charAt(0) != word.charAt(word.length() - 1)) {
        return false;
    }
    else {
        return isPalindrome(word.substring(1, word.length() - 1));
    }
}

• But there are many sub-strings created for the intermediate recursive calls!
• A recursive helper method that allows you to specify the “range” of substring can help.
public static boolean isPalindrome(String word) {
    isPalindromeHelper(0, word.length() - 1, word);
}

private static boolean isPalindromeHelper(int start, int end, String word) {
    if (start >= end) {
        return true;
    } else if (word.charAt(start) != word.charAt(end)) {
        return false;
    } else {
        return isPalindromeHelper(start + 1, end - 1, word);
    }
}
Index (2)

Tree Operation (1.2):
Computing the Depth of a Node

Tree Operation (2.1):
Computing the Height of Tree

Tree Operation (2.2):
Computing the Height of Tree

Exercises (1)

Binary Trees

Binary Trees: Terminology (1)

Binary Trees: Recursive Definition

Binary Trees: Terminology (2)

Background (1): Sum of Geometric Sequence

Binary Trees: Properties (1)

Binary Trees: Properties (2)
Index (3)

Binary Trees: Properties (3)
Binary Trees: Properties (4)
Binary Trees: Properties (5)
Binary Trees: Terminology (3)
Binary Trees: Properties (6)
Binary Trees: Terminology (4)
Binary Trees: Terminology (5)
Binary Trees: Application (1)
Binary Trees: Application (2)
Tree Traversal Algorithms: Definition
Tree Traversal Algorithms: Common Types
Tree Traversal Algorithms: Preorder
Tree Traversal Algorithms: Postorder
Tree Traversal Algorithms: Inorder
Index (4)

Trees in Java: Class Hierarchy
General Tree: Java Interface
Trees in Java: Class Hierarchy
General Tree: Java Abstract Class (1)
General Tree: Java Abstract Class (2)
General Tree: Java Abstract Class (3)
General Tree: Java Abstract Class (4)
General Tree: Java Abstract Class (5)
General Tree: Java Abstract Class (6)
Trees in Java: Class Hierarchy
Binary Tree: Java Interface
Trees in Java: Class Hierarchy
Binary Tree: Java Abstract Class (1)
Index (5)

Binary Tree: Java Abstract Class (2)
Binary Tree: Java Abstract Class (3)
Binary Tree: Java Abstract Class (4)
Binary Tree: Java Abstract Class (5)
Trees in Java: Class Hierarchy
Linked Binary Tree: Linked Object Structure
Linked Binary Tree: Java Class (1)
Linked Binary Tree: Java Class (2)
Linked Binary Tree: Java Class (3)
Linked Binary Tree: Java Class (4)
Linked Binary Tree: Java Class (5)
Linked Binary Tree: Java Class (6)
Linked Binary Tree: Java Class (7)
Linked Binary Tree: Java Class (8)
Linked Binary Tree: Java Class (9)

A Note on Recursive Helper Method (1)

A Note on Recursive Helper Method (2.1)

A Note on Recursive Helper Method (2.2)