Universidad Carlos III de Madrid

Algorithms and Data Structures (ADS)
Bachelor in Informatics Engineering
Computer Science Department

Binary Trees.

Authors: Isabel Segura-Bedmar

April 2011
Departamento de Informtica,
Laboratorio de Bases de Datos Avanzadas (LaBDA)
http://labda.inf.uc3m.es/

Binary Trees

Most of the information has been sourced from the books [1, 2]

Definition.

The order of a **binary** tree is always 2, that is, every node has at most two children (usually called *left child* and *right child*). Given an internal node v, the subtree rooted by its left child is called *left subtree* and the one rooted by its right child is called *right subtree*. For example, Figure 1 shows a simple binary tree rooted with a node whose value is 2. The left child of the root node is the subtree rooted by the node whose value is 7, while the right child is the subtree rootd by the node with value 5.

Besides, a binary tree can be defined in a recursive fashion. A binary tree is either empty, or is made of a single node, whose the left and right children are binary trees too.

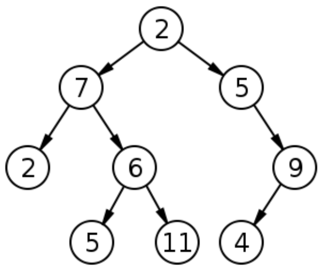


Figure 1: Example of a simply binary tree.

A binary tree is ordered if for every node v, the values in its left subtree are less than the value in v, and all values in its right subtree. The binary tree shown in Figure 1 is not ordered, but Figure 2 shows an ordered binary tree because all their nodes fulfill alphabetical order.

A *full* binary tree (also called *proper*) is a tree in which every internal node has two children (see Figure 3). The trees shown in Figures 1 and 2 are not full binary trees, because both have internal nodes with only one child.

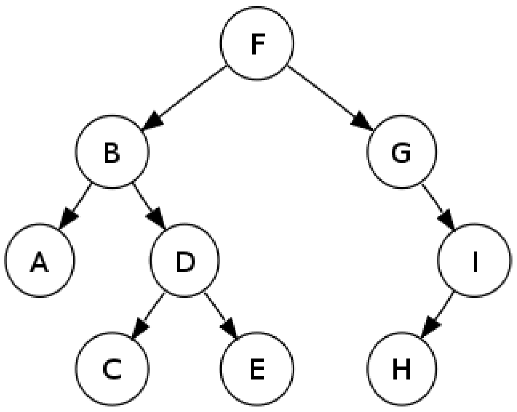


Figure 2: Example of an ordered binary tree.

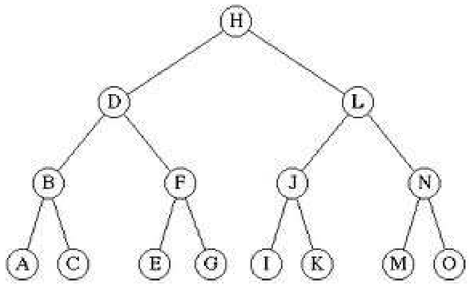


Figure 3: Example of a full (proper) binary tree.

The full binary trees are often used to represent decision trees. These are models to assist the decision maker in finding the

Full binary trees are often used to represent decision trees. A decision tree is a model of decision to assist the decision maker in finding the option depending on whether the answer is ’Yes’ or ’No’. Figure 4 illustrates a decision tree that allows to order three elements A, B and C. The internal nodes represent comparison operations and the leaves represent the possible outcomes.

Besides, binary trees can be useful to represent arithmetic expressions. The internal nodes are operators, while the external nodes represent variables or constants. The value of an internal node can be calculated by applying its operation to the values of its children. Figures 5 and 6 show two binary trees representing arithmetic expresion. The former three has variables and constants in its leaves, but the second one only has constants. The value associated with each internal node is shown next to each node. For example, the root has the value 4 (the final outcome of the arithmetic expression).

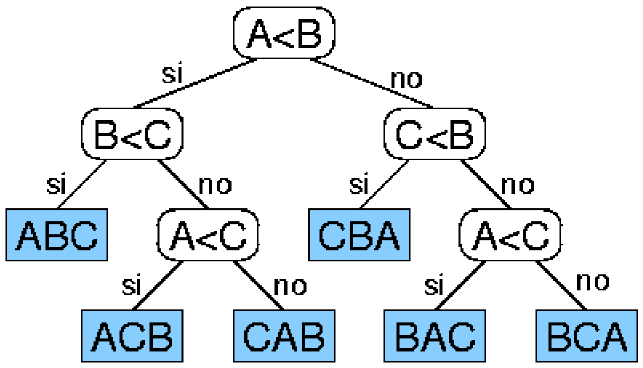


Figure 4: Example of a decision tree to order three elements A, B and C.

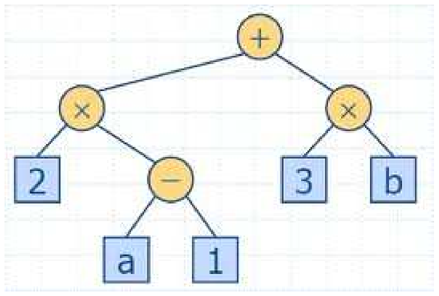


Figure 5: Example of a binary tree to represent the arithmetic expression: 2 x (a-1) + 3 x b.

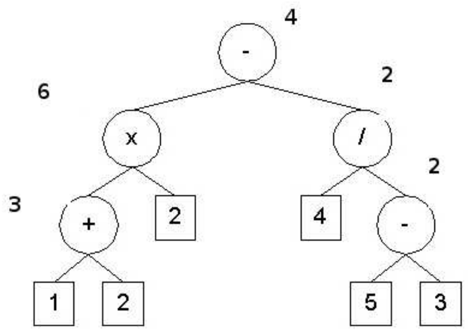


Figure 6: Example of a binary tree to represent the arithmetic expression: (1+2) x 2 - 4 / (5-3).

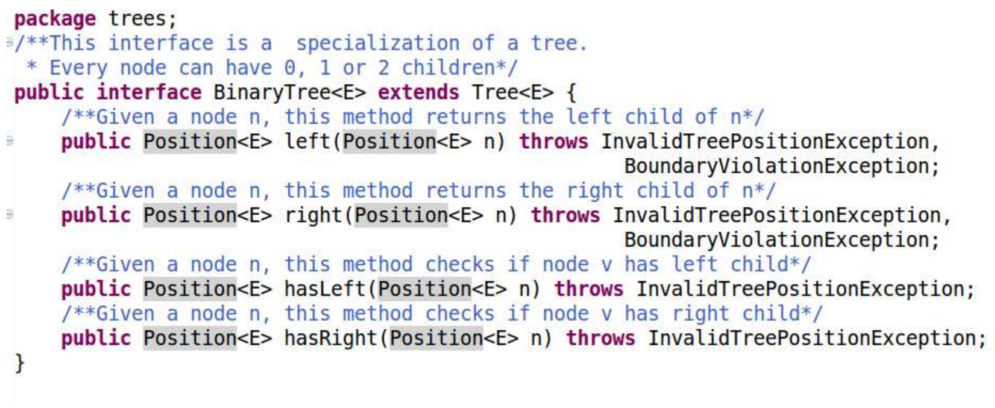
The binary tree ADT

A binary tree is a specialization of a tree. Also, the definition of a binary tree requires at least the following methods:

- hasLeft(v): returns true if the node v has left child.
- hasRight(v): returns true if the node v has right child.
- left(v): returns the left child of the node v. If v does not have left child, an error occurs.
- right(v): returns the right child of the node v. If v does not have left child, an error occurs.

You may want to have an additional field to store its parent

Figure 7 shows an interface for the binary tree ADT. This interface extends the interface Tree(see code fragment in Figure **??**). If the tree is ordered, then the method children(v) returns first the left child followed by the right one.



```java
package trees;
/**This interface is a  specialization of a tree.
 * Every node can have 0, 1 or 2 children*/
public interface BinaryTree<E> extends Tree<E> {
    /**Given a node n, this method returns the left child of n*/
    public Position<E> left(Position<E> n) throws InvalidTreePositionException,
                                                  BoundaryViolationException;
    /**Given a node n, this method returns the right child of n*/
    public Position<E> right(Position<E> n) throws InvalidTreePositionException,
                                                  BoundaryViolationException;
    /**Given a node n, this method checks if node v has left child*/
    public Position<E> hasLeft(Position<E> n) throws InvalidTreePositionException;
    /**Given a node n, this method checks if node v has right child*/
    public Position<E> hasRight(Position<E> n) throws InvalidTreePositionException;
}
```

Figure 7: An interface for the binary tree ADT.

Properties of a binary tree

Firstly, we must define two concepts: **level** (or depth) of a node and **height** a tree.

Given a node v, we can define its level in a recursive fashion, as follows:

- If v is the root of the tree, its level is 0.
- otherwise, its level (or level) is 1 + the level of its parent node.

Figure 8 represents an ordered binary tree and shows the level for each node.

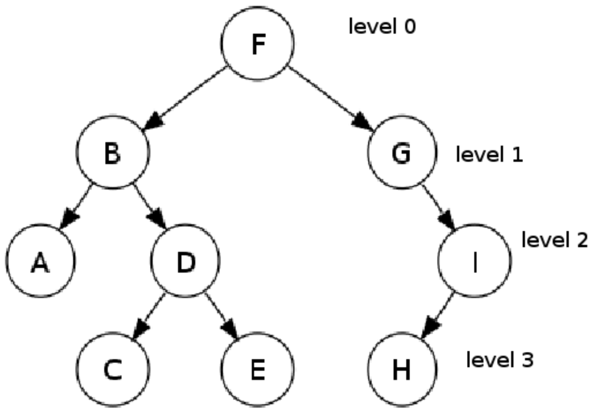
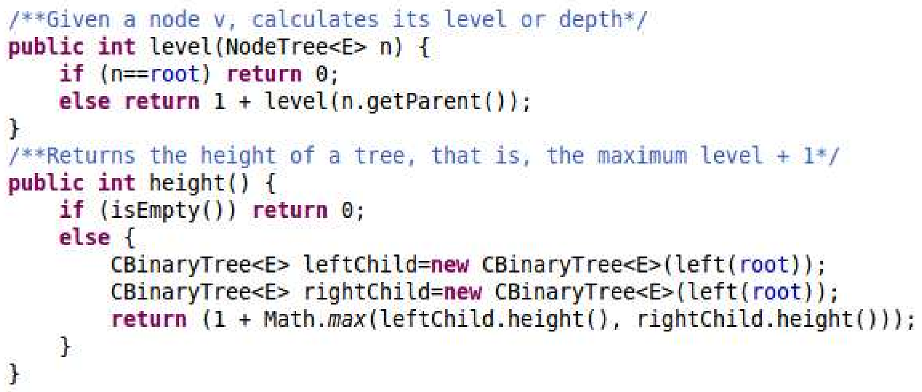


Figure 8: Level (or depth) of a node.

The height of a tree can be defined as 0 if the tree is empty, and otherwise, as 1 + plus the maximum value between the height of the left subtree of its root and the height of the right subtree of its root. Figure 9 shows the methods to calculate these properties of a tree.



```
/**Given a node v, calculates its level or depth*/
public int level(NodeTree<E> n) {
    if (n==root) return 0;
    else return 1 + level(n.getParent());
}
/**Returns the height of a tree, that is, the maximum level + 1*/
public int height() {
    if (isEmpty()) return 0;
    else {
        CBinaryTree<E> leftChild=new CBinaryTree<E>(left(root));
        CBinaryTree<E> rightChild=new CBinaryTree<E>(left(root));
        return (1 + Math.max(leftChild.height(), rightChild.height()));
    }
}
```

Figure 9: Implementation of the methods to obtain the height of a tree and the level of a node.

The level 0 of a nonempty binary tree only has one node (root) (=2^0), the first level has at most two nodes (=2^1), the second level has at most four nodes (=2^2), the third level hast at most eight nodes (=2^3), and so on. You can see that the maximum number of nodes on levels grows exponentially. In general, a binary tree in its leven n has at most 2^n nodes.

Now, you must try to demostrate the following properties:

1. **A full binary tree of height h has $(2^(h+1)-1)$ nodes**.

Let #T denotes the number of nodes of T and #Level$_i$ the number of nodes in the level i. Then,

$$\#T = \#Level_0 + \#Level_1 + \#Level_2 + \ldots + \#Level_h = 2^0 + 2^1 + 2^2 + \ldots + 2^h \quad (1)$$

You can find a tip in the Appendix A: Useful Mathematical Facts [1] to solve this equation. In particular, you should use the proposition A.14:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \quad (2)$$

$$\#T = 2^0 + 2^1 + 2^2 + \ldots + 2^h = \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1 \quad (3)$$

2. **In an full binary tree, the numer of external nodes (leaves) is 1 more than the number of internal nodes**.

Let e, i denotes the number of leaves and the number of internal nodes in the tree T, respectively. So, we must demostrate that:

$$e = i + 1 \quad (4)$$

To justify this property, we are applying the induction principle (please, you must study and look up Section 4.3.3 Induction and Loop Inva riants in the book [1]). Concisely, given a statement P(n), the induction principle proves its correctness for n=1 (2,3). Then, this principle assumes that the statement is held for an arbitrary n, and tries to prove for n+1.

You can easily check (by drawing trees) the following statements.

- If the tree has 1 nodes, the property 2 is satisfied.
- If the tree has 3 nodes, the property 2 is satisfied.
- If the tree has 5 nodes, the property 2 is satisfied.

Note tha the tree neve has a pair number of nodes because it is a full binary tree. Now, we assume true for a tree with n nodes ((a) e$_n$ = i$_n$ + 1). What does it happen if we add new external nodes?. We cannot add only one leave becuase this violate the property of full binary tree. So, we must add two nodes. It is clear that a leave turns into an internal node in the new tree, that is, (b) e$_{n+2}$=e$_n$-1 and (c) i$_{n+2}$=i$_n$+1:

$$e_{n+2} =_{(b)} (e_n - 1) + 2 = e_n + 1 =_{(a)} i_n + 1 + 1 =_{(c)} i_{n+2} + 1 \quad (5)$$

3. **The number of leaves satisfies the following equation**:

$$h \leq e \leq 2^h \tag{6}$$

You can use the induction principle to prove $h \leq e$.

- if $h = 1$, it is obvious that $1 \leq e$, becuase $n = e = 1$.

- Now, we assume that $h \leq e$, and we must prove for $h + 1 \leq e_{new}$, that, it is the number of leaves in the new tree. If we increase the height of the tree, we must add at leat two nodes and a leave turns into internal node, therefore, (a) $e_{new} = (e - 1) + 2$

$$h + 1 \leq_{h \leq e} e + 1 = (e - 1) + 2 =_{(a)} e_{new} \tag{7}$$

Now, we will demostrate the second part of the equation ($e \leq 2^h$).

$$n = e + i =_{2ndproperty} e + (e - 1) = 2e - 1 =_{1stproperty} 2^{h+1} - 1 \tag{8}$$

$$2e = 2^{h+1} \Leftarrow e = 2^h \tag{9}$$

For every nonfull tree, it is obvious that $e \leq 2^h$

The following properties can be proved based on the three previous properties. Please, try yourself!!!:

4. **The number of internal nodes satisfies the following equation:**

$$h \leq i \leq 2^h - 1 \tag{10}$$

5. **The total number of nodes satisfies the following equation:**

$$2h + 1 \leq n \leq 2^{h+1} - 1 \tag{11}$$

6. **The height (h) of a tree satisfies the following equation:**

$$log_2(n + 1) - 1 \leq h \leq \frac{n - 1}{2} \tag{12}$$

7. **The height (h) of a tree satisfies the following equation:**

$$log_2(e) \leq h \leq e - 1 \tag{13}$$

Linked Structure-based implementation of binary tree

To implement a binary tree, we can use a linked structured of nodes to represent each node of the tree. Each node (see Figure 10) the linked structured has the following fields:

- the element stored in the node.
- a reference to its parent. If the node is the root, then this filed is null.
- a reference to its left child. If the node does not have left child, this field is null.
- a reference to its right child. If the node does not have right child, this field is null.

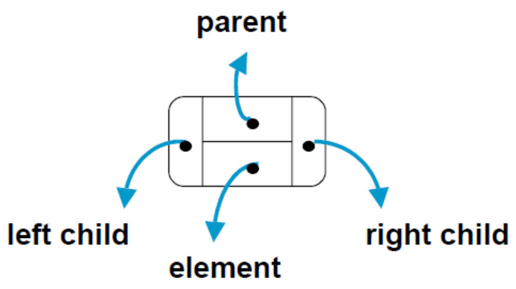


Figure 10: A node of a linked structure to represent a node of a binary tree.

Figure11 shows the representation of a binary tree (whose root is A) by a linked list of nodes. Figure 12 shows an interface for representing binary tree nodes. It has methods to set and return the parent, the left child, the right child and the element stored at a node. Figure 13 shows the clas BTNode which contains four fields: an element, its parent, its left child and its right child. Also, this class implements the methods defined in the interface BTPosition.

Now, we are defining the class *LinkedBinaryTree* that stores a reference to the root of the tree and also the total number of nodes (size). This class implements the interface BinaryTree (see Figure 7). Also, the class has a constructor without arguments that returns an emtpy tree. Besides, the additional methods are defined:

- *addRoot(e)*: creates a new node for storing the element e. This new node is the root of the tree. The method only works when the tree is empty, otherwise an error will occur.[1]

[1]If we define a constructor with a node as input argument that sets the root as this node, then we do not need the method addRoot.

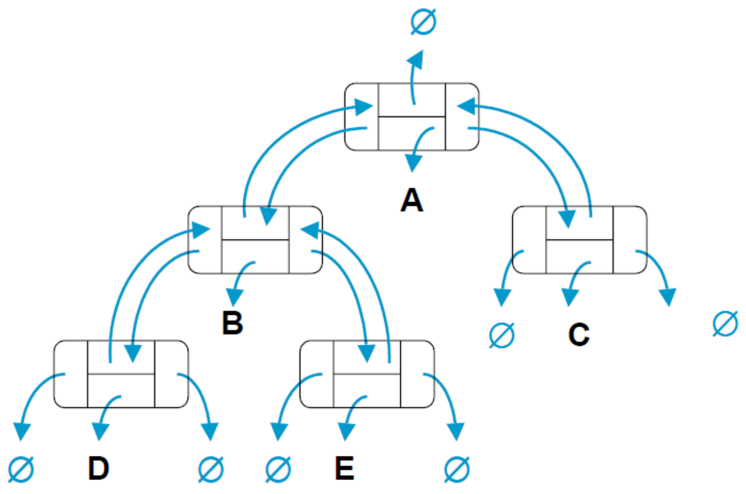


Figure 11: A linked structure for representing a binary tree.

- *addLeft(n,e)*: creates a new node for storing the element e and adds this new node as the left child of the node n. If the n already has a left child, an error will occur.

- *addRight(n,e)*: creates a new node for storing the element e and adds this new node as the right child of the node n. If the n already has a right child, an error will occur.

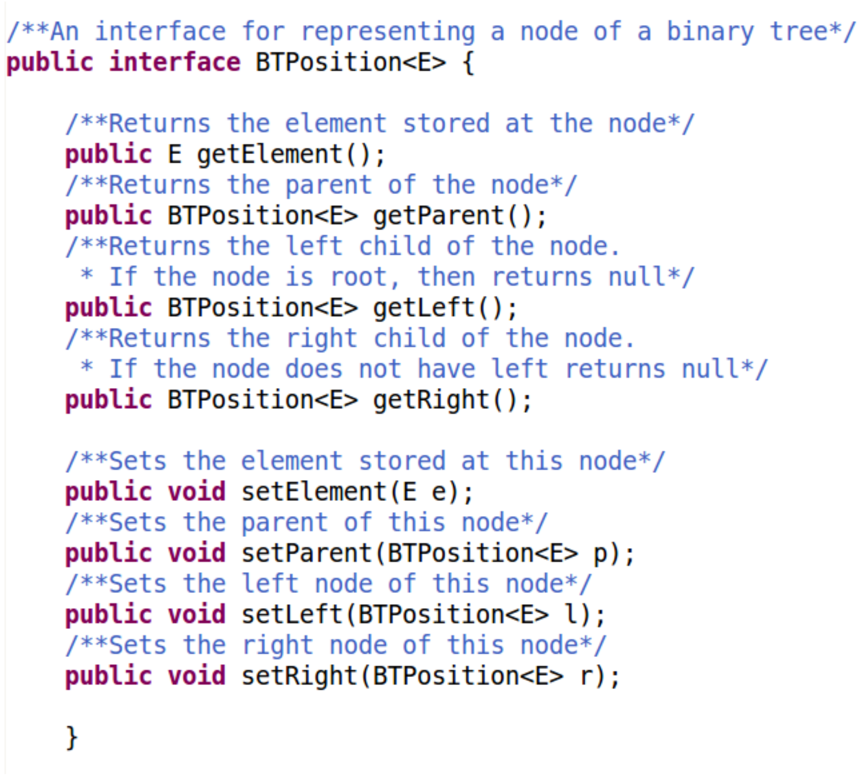
- *attach(n,T_1,T_2)*: attaches the binary trees T_1,T_2 as left and right subtrees of the leave n, respectively. If n is not an external node, then an error will occur.

- *height()*: returns the height of the tree.

These additional methods allow us to build a binary tree by creating the root using the method addRoot and adding the left and right children using the methods addLeft and addRight, repeatedly. You can find the implementation of the class `LinkedBinaryTree.pdf`LinkedBinaryTree. Please, add code to the main method to create a binary tree and test the class. For example, you can try to build the previous examples presented for this section.

Table **??** summarizes the computational complexity for each method in a linked structure implementation of a binary tree.

The method *size()* takes O(1) time because it only uses the instance variable *size*. The methods *isEmpty(),isRoot(),getRoot()* take O(1) time since they only access the instance variable *root*. The methods *hasLeft(), left(), hasRight(), right()* take O(1) because they only access the instance variables left,right,parent (of the BTNode class), respectively. Likewise, the methods sibling(), isLeave(),



```
/**An interface for representing a node of a binary tree*/
public interface BTPosition<E> {

    /**Returns the element stored at the node*/
    public E getElement();
    /**Returns the parent of the node*/
    public BTPosition<E> getParent();
    /**Returns the left child of the node.
     * If the node is root, then returns null*/
    public BTPosition<E> getLeft();
    /**Returns the right child of the node.
     * If the node does not have left returns null*/
    public BTPosition<E> getRight();

    /**Sets the element stored at this node*/
    public void setElement(E e);
    /**Sets the parent of this node*/
    public void setParent(BTPosition<E> p);
    /**Sets the left node of this node*/
    public void setLeft(BTPosition<E> l);
    /**Sets the right node of this node*/
    public void setRight(BTPosition<E> r);

    }
```

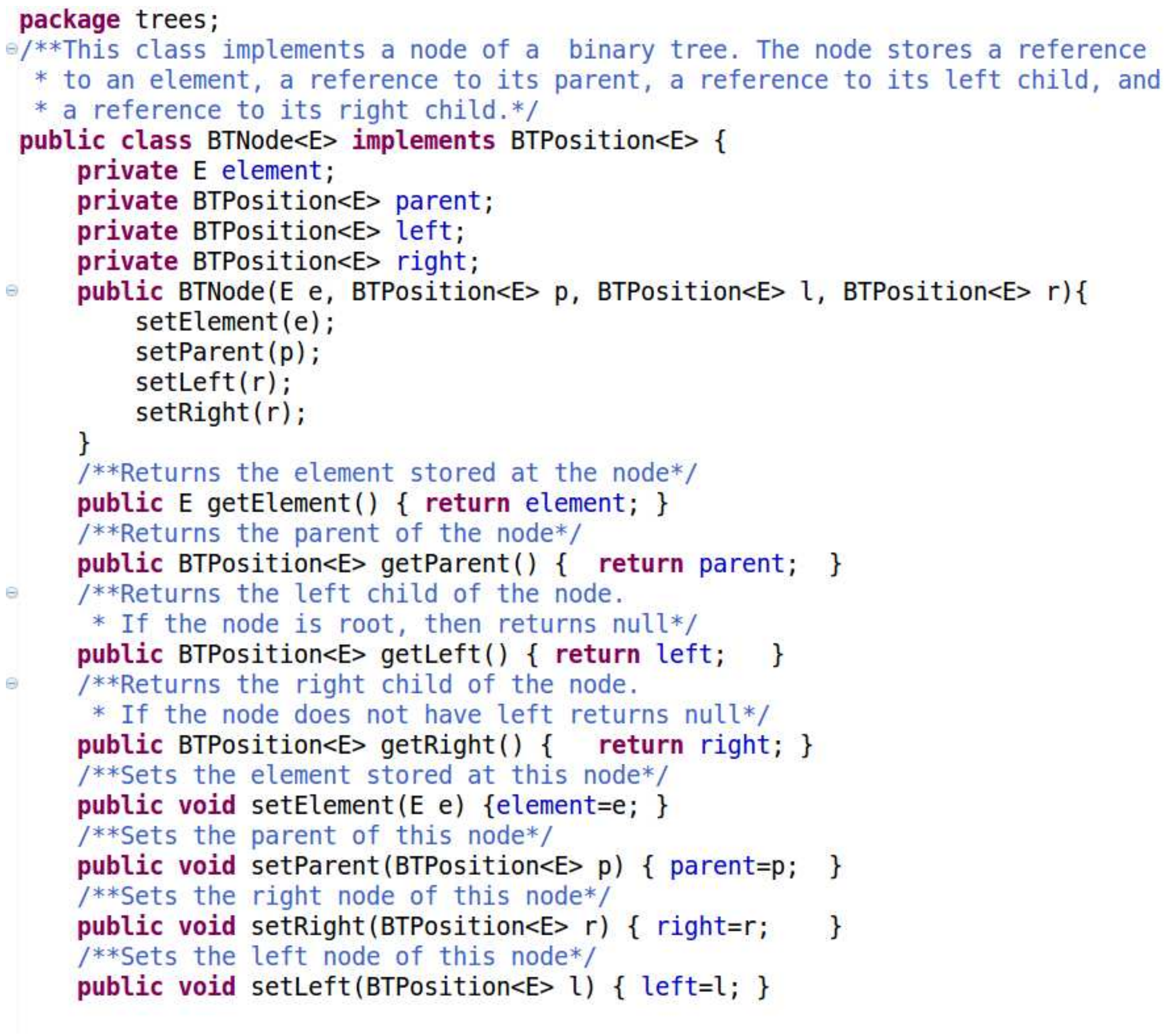
Figure 12: An interface to represent a node of a binary tree.

isInternal() takes O(1) time becuase they only access instances variables. Since the method *children()* just need to access two instance variables (left and right children of a give node), it only takes O(1) time. The method *positions* uses an recursive method *preorderPositions* that traverses the tree and stores its nodes in a list. Thus, *positions* takes O(n) time. Likewise, *iterator* also takes O(n) since it uses the method *positions*. The methods *replace* and *addRoot* takes O(1) time because they access and use one node. The methods *insertLeft*, *insertRight* and *remove* takes O(1) time because they access and modify a constant number of nodes.

Array-based implementation of a binary tree

Another alternative to implement a binary tree is to store the nodes of the tree in an array. The root of the tree is stored in the first position in the array, its left child in the second position, its right child in the third position, and so on (see Figure 14).

Since we cannot know the maximum size that the tree may reach, the best



```
package trees;
/**This class implements a node of a  binary tree. The node stores a reference
 * to an element, a reference to its parent, a reference to its left child, and
 * a reference to its right child.*/
public class BTNode<E> implements BTPosition<E> {
    private E element;
    private BTPosition<E> parent;
    private BTPosition<E> left;
    private BTPosition<E> right;
    public BTNode(E e, BTPosition<E> p, BTPosition<E> l, BTPosition<E> r){
        setElement(e);
        setParent(p);
        setLeft(r);
        setRight(r);
    }
    /**Returns the element stored at the node*/
    public E getElement() { return element; }
    /**Returns the parent of the node*/
    public BTPosition<E> getParent() {  return parent;  }
    /**Returns the left child of the node.
     * If the node is root, then returns null*/
    public BTPosition<E> getLeft() { return left;   }
    /**Returns the right child of the node.
     * If the node does not have left returns null*/
    public BTPosition<E> getRight() {   return right; }
    /**Sets the element stored at this node*/
    public void setElement(E e) {element=e; }
    /**Sets the parent of this node*/
    public void setParent(BTPosition<E> p) { parent=p;  }
    /**Sets the right node of this node*/
    public void setRight(BTPosition<E> r) { right=r;    }
    /**Sets the left node of this node*/
    public void setLeft(BTPosition<E> l) { left=l; }
```

Figure 13: A class for implementing binary tree nodes.

option is to use an ArrayList to store its nodes (we do not recommend to use the zero-positon of the arraylist).

It is possible to know the position of a given node from the position of its parent. For every node, Figure 14 clearly demostrates the following claims:

- The position of its left child in the arraylist is: *pos(left)=2*pos(node)*.
- The position of its right child in the arraylist is: *pos(right)=2*pos(node)+1*

Now, you should give the arraylist-based implementation of a binary tree. Then, you will have to estimate the running times of its main methods.

Methods	Time
isEmtpy(), isRoot(), size()	O(1)
hasLeft(), getLeft(), hasRight(), getRight, parent(), sibling	O(1)
isLeave(), isInternal()	O(1)
positions(), iterator()	O(n)
replace(), addRoot()	O(1)
insertLeft(), insertRight(), remove()	O(1)

Table 1: Performance of an linked structure-based implementation of a binary tree

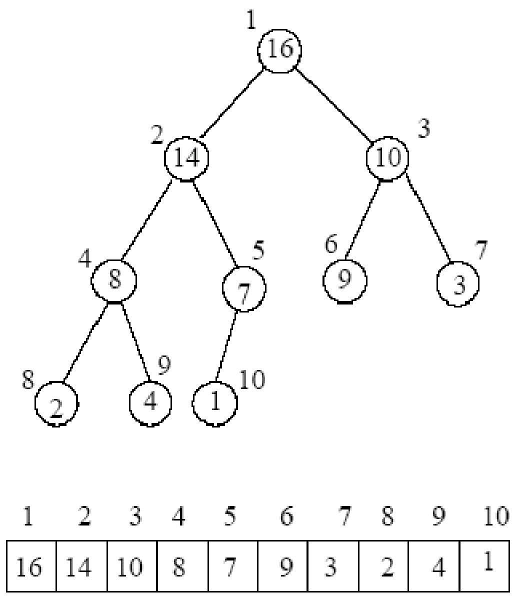


Figure 14: Running times for a binary tree implemented with an linked structure

Methods	Time
isEmtpy(), isRoot(), size()	O(1)
hasLeft(), getLeft(), hasRight(), getRight, parent()	O(1)
isLeave(), isInternal()	O(1)
positions(), iterator()	O(n)
replace(), addRoot()	O(1)
insertLeft(), insertRight(), remove()	O(1)

Table 2: Running times for a binary tree implemented with an arraylist

Bibliography

[1] M. Goodrich and R. Tamassia, *Data structures and algorithms in Java*. Wiley-India, 2009.

[2] M. Weiss, *Data structures and problem solving using Java*. Addison Wesley Publishing Company, 2002.