1 Implementation of the Binary Tree ADT

1.1 Implementation using Arrays

Representing linear structures (lists, stacks, etc.) using arrays was a quite natural operation, considering that the array itself is an inherently linear organization (a sequence of variables).

Trees are not linear. Thus representing them using arrays requires a proper mapping from the tree structure into a linear organization. In general representing trees using arrays is feasible (without excessive complications) only if we know a priori what is the maximum number of edges coming out of each node. Binary trees fall in this class, as we know that each node is going to have at most two successors.

Consider the tree in Figure 1(i). The nodes in the tree has been numbered consecutively, starting from 1 (root), 2 and 3 have been assigned to the children of the root, 4, . . . , 7 have been used for the grandchildren of the root and so on. The intuition, as shown in Figure 1(i), is that since

- the first $k$ levels of the tree can contain at most $1 + 2 + 4 + \ldots + 2^{k-1}$ nodes, i.e., $2^k - 1$ nodes;
- the level $k$ of the tree will contain at most $2^k$ nodes;

then we can conclude that by assigning a consecutive number to each node, moving down level by level, the nodes at level $k$ will use the numbers from $2^k$ to $2^{k+1} - 1$.

This is suggesting an easy idea for representing trees as arrays; we consider the above numbering scheme, and if a node is assigned node $i$, then we just simply store that node in the location $a[j]$ of the array. This is show in Figure 1(ii).

Thus, if we are positioned in a node which is lying in position $i$ of the array, then we can easily identify:

- the left child will be in position $2 \times i$;
- the right child will be in position $2 \times i + 1$;
- its parent (if the node is not the root) is in position $i/2$;
- the root is in position 1

We assume that if a node does not exist, then the corresponding entry in the array is filled with a default value (e.g., $-1$).

The major problem is that the array required to store even a relatively small tree can be very large. If we consider the tree containing $k$ nodes $n_1, n_2, \ldots, n_k$ organized as follows: $n_1$ is the root, $n_2$ is the right child of $n_1$, $n_3$ is the right child of $n_2$, . . . , $n_k$ is the right child of $n_{k-1}$, then we can easily see that we will need an array of size $2^k - 1$ (e.g., for $k = 10$ we need an array with $2^{10} - 1 = 1023$ positions—thus we have 1023 positions used to store a tree which contains only 10 nodes...).
1.2 An Application

Consider the problem of managing a distribution network. We have a source which produces a certain resource (e.g., petroleum). A network distributes the resource from the source to a number of destinations (points of consumption). It is common that the structure of the distribution network is a tree structure, where the root is the source, the leaves are the consumption points, and the intermediate nodes are possible substations.

Along the path, the resource may experience loss or degradations of one of its characteristics. E.g., if the network distributes petroleum, then it is possible to have pressure drop along the lines. Between the source and the consumption points we can tolerate only a certain degree of degradation (called tolerance). To guarantee that degradation does not exceed the tolerance, we can place signal boosters in some of the substations of the network. The booster will restore the original value of the pressure, eliminating the degradation accumulated to that point.

Consider for example the network in Figure 2.

The root (p) is the source. The leaves (w,x,y,z) are the consumption points. All other nodes are substations.
A substation may or may not contain a booster. The numbers on the edges are used to measure the degradation along that segment of the network. Thus, going from q to t we will experience a degradation of 2 points. Degradation is an additive property. Thus, going from p to w we will experience a total degradation of 3 + 1 = 4.

The problem is to determine the best placement of boosters at the substations. We want to place the minimum number of boosters that will guarantee that the resource will get to each consumption point with a degradation which is not superior than the tolerance.

In the example, if the tolerance is 4, if we do not put any booster in the path from the source to w then we will violate our safety property, because on such path we have a total degradation of 1 + 2 + 3 = 6 which is greater than the tolerance (4). If we place a booster in the substation s, then that path will become safe, since up to s we have a total degradation of 3 (below tolerance), and at s all the degradation is removed by the booster; this allows the resource to get to the consumption point w with a degradation of 3 (due to the degradation between s and w).

The solution strategy is as follows:

• let us denote with $\text{degradeFromParent}(i)$ the degradation which occurs between node $i$ and its parent. In the example we have that $\text{degradeFromParent}(w)=3$, $\text{degradeFromParent}(s)=2$, and $\text{degradeFromParent}(q)=1$.

• let us denote with $\text{degradeToLeaf}(i)$ the maximum degradation experience between node $i$ and any of its leaf successors. In the example, we have that $\text{degradeToLeaf}(q)=5$ and $\text{degradeToLeaf}(r)=4$.

In general we can compute the $\text{degradeToLeaf}$ as follows:

$$\text{degradeToLeaf}(i) = \max_{j \text{ child of } i} \{ \text{degradeToLeaf}(j) + \text{degradeFromParent}(j) \}$$

The value of this function for each node in the tree can be easily computed by starting from the leaves (whose value is zero) and then progressively moving towards the root.

If during the computation we determine that $\text{degradeToLeaf}(j) + \text{degradeFromParent}(j) > \text{tolerance}$

then we need to place a booster at node $j$.

In the example, when we compute $\text{degradeToLeaf}(s) + \text{degradeFromParent}(s)$

we obtain 5 which violates a tolerance of 4. If we place a booster at node s, then the degradation above s disappears and we will get a $\text{degradeToLeaf}(q) = 3$.

Thus the pseudocode for computing the $\text{degradeToLeaf}$ values and placing boosters is as follows:

```plaintext
degrateToLeaf(i) = 0;
for each j child of i do
  if (degradeToLeaf(j) + degradeFromParent(j) > tolerance) then
    place a booster at j
    degradeToLeaf(i) = max(degradeToLeaf(i), degradeFromParent(j));
  else
    degradeToLeaf(i) = max(degradeToLeaf(i), degradeFromParent(j)+degradeToLeaf(j));
```

It is possible to prove that this produces the optimal placement of boosters in the network.

1.3 Traversing Trees

A typical problem is that of traversing a binary tree, i.e., scanning all the nodes of the tree in a predefined order. We can start by distinguishing two forms of tree traversal:

1. Depth-first traversal: the processing proceeds along a path from the root to one child to the most distant descendent of that first child before processing a second child. I.e., all successors of a child are processed before moving to another child.
2. *Breadth-first traversal*: the processing proceeds horizontally from the root to all of its children, then to its children’s children and so forth until all nodes have been processed.

1.3.1 Depth-First Traversal

There are three typical orders for (depth-first) traversing a tree:

- *PreOrder*;
- *PostOrder*;
- *InOrder*

Consider the example of the Figure 3. The tree contains the description of an arithmetic expression. Let us consider the binary tree to have base type `char` (i.e., the value in each node of the tree is a single character).

**Figure 3: Expression Tree**

**PreOrder**: Visiting a tree in preorder involves exploring the tree as follows:

1. first of all visit the root of the tree (and read its content);
2. visit in PreOrder the left subtree of the root;
3. visit in PreOrder the right subtree of the root;

In the example of Figure 3, a PreOrder visit will thus produce the following sequence of values:

```
/ - ∧ b 2 * * 4 a c * 2 a
```

This visit can be implemented as follows, as part of the implementation using linked lists:

```java
void PreOrderTraverse (BinaryTree t)
{
    if (!t.emptyTree())
    {
        System.out.print(“ ’ ”+t.getRoot()+”’ ”);
        PreOrderTraverse(t.getLeft());
        PreOrderTraverse(t.getRight());
    }
}
```
**PostOrder:** Visiting a tree in postorder involves exploring the tree as follows:

1. first visit the whole left subtree in PostOrder;
2. then visit the whole right subtree in PostOrder;
3. the visit the root of the tree;

In the example of Figure 3, a PostOrder visit will thus produce the following sequence of values:

\[ b 2 \land 4 a * c * - 2 a * / \]

This visit can be implemented as follows, as part of the implementation using linked lists:

```java
void PostOrderTraverse (BinaryTree t) {
    if (!t.emptyTree()) {
        PostOrderTraverse(t.getLeft());
        PostOrderTraverse(t.getRight());
        System.out.print('' +t.getRoot()+'' );
    }
}
```

**InOrder:** Visiting a tree in inorder involves exploring the tree as follows:

1. first visit the whole left subtree in InOrder;
2. then visit the root of the tree;
3. then visit the whole right subtree in InOrder;

In the example of Figure 3, a InOrder visit will thus produce the following sequence of values:

\[ b \land 2 - 4 * a * c / 2 * a \]

This visit can be implemented as follows, as part of the implementation using linked lists:

```java
void InOrderTraverse (BinaryTree t) {
    if (!t.emptyTree()) {
        InOrderTraverse(t.getLeft());
        System.out.print('' +t.getRoot()+'' );
        InOrderTraverse(t.getRight());
    }
}
```

1.3.2 Breadth-First Traversal

In the breadth-first traversal of a binary tree, we process all of the children of a node before proceeding with the next level. In other words, given a root at level \( n \), we process all nodes at level \( n \) before proceeding with the nodes at level \( n + 1 \).

While depth-first traversal are made according to the recursive structure of a tree (i.e., a tree is a root with two other trees attached below), breadth-first is instead ignoring this structuring and cutting the tree horizontally by levels. This implies that recursive solutions are not going to help in writing a breadth-first procedure. The order in which nodes are considered matches well with the FIFO order provided by a queue (in which nodes at level \( n \) are inserted in the queue before any node of level \( n + 1 \); as consequence all nodes of level \( n \) are going to be encountered while dequeueing the queue before encountering any node of level \( n + 1 \)). The breadth-first traversal of the tree in figure 3 will produce
The following is a procedure for performing breadth-first traversal (it assumes the availability of a queue ADT—the base type for the queue is BinaryTree):

```java
void BreadthFirst (BinaryTree t)
{
    Queue q;
    q = new Queue();
    if (t.emptyTree())
        return;
    else
        q.enqueue(t);
    while (! q.emptyQueue())
    {
        t = q.front();
        q.dequeue();
        System.out.print('' + t.getRoot());
        if (! t.getLeft().emptyTree())
            q.enqueue(t.getLeft());
        if (! t.getRight().emptyTree())
            q.enqueue(t.getRight());
    }
}
```

### 2 Binary Search Trees

A binary search tree is a data structure which is used to store information identified by unique keys. Each piece of information is uniquely associated to a key. You can think about keys as non-negative integer numbers.

The problem is to maintain the set of data in such a way to allow fast retrieval of information; given a key we would like to quickly discover whether such key has been used, and in this case we want to extract the corresponding piece of information.

Binary search trees are commonly used to solve this problem. The keys are stored in the nodes of a binary tree (we can assume the existence of an external table which maps each key to the corresponding data). Keys are inserted in the tree according to the following rule: to insert the key $k$ in the tree (see also Figure 4(i)):

```java
public void insertBST (int key)
{
    BinaryTree r, parent;
    Node n;
    n = new Node(key);
    if (root == null)
    {
        root = n;
    }
    else
    {
        r = this;
        while (! r.emptyTree())
        {
            parent = r;
            if (key == r.getRoot())
                return;
            if (key < r.getRoot())
                r = r.getLeft();
            else
                r = r.getRight();
        }
        parent.left = n;
        n.parent = parent;
    }
}
```
return;
if (key < r.getRoot())
    r = r.getLeft();
if (key > r.getRoot())
    r = r.getRight();
}  
if (key < parent.getRoot())
    parent.insertNode(key,0);
else
    parent.insertNode(key,1);
}

Thus, to summarize, for each node $N$ (containing the key $k$) in a binary search tree:

- all the keys in the left subtree of $N$ have value smaller than $k$;
- all the keys in the right subtree of $N$ have value bigger than $k$.

Figure 4(ii) illustrates an example of search tree.

![Binary Search Tree Diagram](image)

20, 10, 25, 21, 5, 15, 31, 17

Figure 4: Binary Search Tree

In order to search a key in the tree, we need to move left or right depending on the result of the comparison between the key and the root of the current subtree.

```java
void Search(int Key) {
    if (emptyTree())
        System.out.println('Key not present');
    else
        if (root.getData == Key)
            System.out.println('FOUND');
```
else
    if (root.getData() > Key)
        getLeft().Search(Key);
    else
        getRight().Search(Key);
}

The more complex operation on binary search trees is the deletion of a key. Since we do not place any restrictions on the insertion of elements (any key can be inserted in the tree), then we should also not place any restriction on which key can be removed. This may potentially lead to complications, since we may be trying to remove a node which has successors, thus leaving a “hole” in the tree. E.g., if we try to remove node 15 from the tree in Figure 4, we need to make sure that we are not loosing access to node 17.

There are four possible cases which may occur:

1. the node to be removed does not have any child; in this case the node can be safely removed without endangering access to other nodes in the tree; we should only make sure that the connection between the node and its parent is removed (e.g., set the leftChild or rightChild of parent to null);
2. if the node has a right subtree but no left subtree, then we can simply attach the right subtree to the node’s parent (see Figure 5(i)).
3. if the node has a left subtree but no right subtree, then we can simply attach the left subtree to the node’s parent;
4. if the node to be deleted has both left and right subtrees, then the simple way to proceed is the following (see also Figure 5(ii)):
   - find the largest node in the left subtree of the node to be deleted (in Figure 5(ii) 20 is the maximum value in the left subtree of 23);
   - copy such value in the node to be delete (in the example, 20 is written in the root of the tree);
   - delete the largest node from the left subtree (in the example, remove the node containing 20 from the subtree rooted in 18);

Note that the last step is a recursive call to the procedure which deletes nodes from a binary search tree. Nevertheless, in this recursive case we are guaranteed that the node to be removed does not have a right subtree (i.e., it will be either case 1 or case 3 above).

2.1 Cost of Searching
If we consider searching one key in a given tree:

Best Case: The best case is the one in which we search for a given key and this key is present in the root of the tree. One comparision is sufficient to produce the result.

Worst Case: the tree is completely unbalanced and is reduced to a straight sequence of nodes. In this case, if there are n nodes in the tree, then we may take up to n comparisons to produce an answer.

What about the case in which we have a given tree and we can make any arbitrary number of queries in it? What is the best case?

It turns out that the best case is the one in which all the paths in the tree have the same length. This occurs if the tree is a complete tree (each internal node has exactly two successors, and the leaves belongs all to the same level). In this case, it is easy to observe that each search will take up to \( \log n \) comparisons, if \( n \) is the number of nodes in the tree.
2.2 Binary Search Tree ADT

2.2.1 Specification

Set of values: the set of all possible values is represented by the set of all possible binary search trees over a given domain.

The content of each node is a non-negative integer number representing the value of the key. Keys are stored in the tree according to the conditions described above.

Operations:

- Constructors: the only constructor needed is the one to create a new object of type binary search tree. Initially the tree created is empty.

  CreateBSTree
  - preconditions: none
  - postconditions: returns a new object of type binary tree containing, as value, the empty tree (i.e., the tree containing zero elements);

- Inspectors: the following inspectors are included in the type specification:

  1. emptyBSTree used to verify whether a given tree is empty or not
     - preconditions: operates on the current tree
     - postconditions: produces true (i.e., 1 in C) if the tree is empty, false (i.e., 0 in C) otherwise
  2. equals used to verify whether two trees have the same value (i.e., they contain the same elements and in the same order).
     - preconditions: receives one tree as input
     - postconditions: returns true if the input tree is identical to the current tree (i.e., they contain the same elements with the same structure), false otherwise.
  3. findKey determines whether a key is present in the search tree
     - preconditions: receives a key k
     - postconditions: it returns the subtree rooted in the searched key (empty tree if the key is not in the tree)
• **Modifiers**: the following modifiers are included

1. **insertKey** inserts a new key in the tree;
   - preconditions: receives in input a key
   - postconditions: inserts the key in the tree according to the rule of search trees;

2. **deleteKey** remove a key from the tree
   - preconditions: receives in input a key
   - postconditions: deletes the key from the tree;

### 2.2.2 Implementation using Linked Structures

We are going to develop an implementation of this ADT using linked structures. We assume that we are going to use this library as an alternative to hash tables to store data. This means that each node in the tree is going to contain two things: a key (an integer number) and a data (in this example we will assume that the data is of type String, but any type could be used here). We will use the functionalities of binary search trees described earlier to manage the keys in the tree. Finding a key will imply giving a key and return the associated data.

We start by modifying the class for Node to accommodate the fact that the nodes are used for encoding binary search trees. The class is similar to the class used earlier, with the exception that the children of a node are going to be binary search trees.

```java
public class Node {
    // data members (private)
    private int key; // the key stored in the node
    private BSTree left; // left child
    private BSTree right; // right child

    // OPERATIONS

    // CONSTRUCTOR
    /***
     * Creates a new node, storing a given data
     * the new node has no children to start with
     *
     * @param d the character to be stored in the node
     ***/
    public Node(int d) {
        key = d;
        left = new BSTree();
        right = new BSTree();
    }

    // INSPECTORS

    /***
     * Read the data stored in the node
     *
     * @return the character stored in the node
     ***/
    public int getData() {
    }
}
```
return key;
}

/***
 * Read the left child of the node
 * @return the reference to the left child
 ***/
public BSTree getLeftChild()
{
    return left;
}

/***
 * Read the right child of the node
 * @return the reference to the right child
 ***/
public BSTree getRightChild()
{
    return right;
}

// MODIFIERS

/***
 * Modify the data part of the node
 * @param d the new integer to be stored in the node
 ***/
public void setData(int d)
{
    key = d;
}

/***
 * Modify the left child part of the node
 * @param l the new left child
 ***/
public void setLeftChild(BSTree l)
{
    left = l;
}

/***
 * Modify the right child part of the node
 * @param r the new right child
 ***/
public void setRightChild(BSTree r)
{
right = r;
}
}

Now we can proceed to define the class for the binary search trees. As you will see, some of the operations are actually similar to those we have seen in the case of binary trees; the differences are present in the existence of specialized operations for searching and insertion.

```java
public class BSTree {
    private Node root;

    // CONSTRUCTOR
    public BSTree() {
        // we start from an empty tree -- just call
        // the superclass constructor
        root = null;
    }

    public BSTree(Node n) {
        root = n;
    }

    // INSPECTORS

    public BSTree getLeftBST() {
        return root.getLeftChild();
    }

    public BSTree getRightBST() {
        return root.getRightChild();
    }

    public int getRootBST() {
        return root.getData();
    }

    public boolean emptyTree() {
        return root == null;
    }

    /***
     * finds a key in the binary search tree
     * @param key the key to be searched
     */
```
public BSTree findKey(int key)
{
    BSTree temp;
    if (emptyTree() || key==getRootBST())
        return this;
    if (key < getRootBST())
    {
        temp = getLeftBST();
        return temp.findKey(key);
    }
    else
    {
        temp = getRightBST();
        return temp.findKey(key);
    }
}

public String toString()
{
    if (emptyTree())
        return " ";
    else
        return getLeftBST().toString()+getRootBST()+getRightBST().toString();
}

// MODIFIERS

/**
 * inserts a new node in the binary search tree
 * @param k the key of the new node
 */
public void insertKey(int k)
{
    Node n = new Node(k);
    if (! findKey(k).emptyTree())
    {
        System.out.println("Key Already Present");
        return;
    }
    // if tree is empty, simple insertion
    if (emptyTree())
        root = n;
    else if (k < getRootBST())
{ 
    if (getLeftBST().emptyTree())
        root.setLeftChild(new BSTree(n));
    else
        getLeftBST().insertKey(k);
} 
else
{
    if (getRightBST().emptyTree())
        root.setRightChild(new BSTree(n));
    else
        getRightBST().insertKey(k);
}

/***
* deletes a key from a binary search tree
* @param k key to be removed
* @return the modified tree
***/
public BSTree deleteKey(int k) 
{
    BSTree parent=null;
    BSTree scan = this;

    // if the key is not there, nothing to remove
    if (findKey(k).emptyTree())
        return this;

    // if node to remove is not the root, just remove recursively
    if (k != getRootBST())
    {
        if (k < getRootBST())
            root.setLeftChild(getLeftBST().deleteKey(k));
        else
            root.setRightChild(getRightBST().deleteKey(k));
        return this;
    }

    // Case 1: Remove a leaf
    if (getLeftBST().emptyTree() && getRightBST().emptyTree())
    {
        return null;
    }

    // Case 2: Remove a node with only a left child

else if (getRightBST().emptyTree())
{
    return getLeftBST();
}

    // Case 3: Remove a node with only a right child
else if (getLeftBST().emptyTree())
{
    return getRightBST();
}

    // case 4: Remove node with both left and right children
else
{
    // find the max in the left subtree
    int k1 = getLeftBST().findMax();

        // remove the max from the left subtree
        root.setLeftChild(getLeftBST().deleteKey(k1));

    // store the max in the root
    root.setData(k1);

    return this;
}

private int findMax()
{
    if (getRightBST().emptyTree())
        return getRootBST();
    else
        return getRightBST().findMax();
}

3 Additional Considerations

3.1 Some Definitions

Some additional terminology related to trees:

• a binary tree is complete if all the leaves of the tree are either:
  – all at the same level;
  – the are at two adjacent levels;

and, furthermore, all the leaves on the bottommost level are placed as far to the left as possible. Figure 6(i)
illustrates a complete tree, while 6(ii) and 6(iii) are not complete trees (the first does not have all leaves
pushed to the left, the second does not have all the leaves in two adjacent levels).
• Consider a binary search tree. Is there any form of traversal that will provide you automatically with the sorting of the elements present in the tree? E.g., if we insert, in the following order, 10, 5, 18, 7, 2, 24, 20, 11 in a binary search tree, is there a traversal that will give us 2, 5, 7, 10, 11, 18, 20, 24?