Trees

We can extend the concept of linked data structures (stack, queue) to structure containing nodes with more than one pointer field. Such structure is called a tree. A tree is a collection of nodes connected by directed edges. A tree is a nonlinear data structure, oppose to linked lists, stacks and queues which are linear data structures. A tree can be empty - no nodes. Otherwise, a tree consists of one node called root and zero or one or more subtrees. A tree has following properties:

- one node is distinguished as a root;
- every node (exclude a root) is connected by a directed edge from exactly one other node;

A direction is: parent -> children

Here is a picture of a tree

A is a parent of B, C, D,
B is called a child of A.
on the other hand, B is a parent of E, F, K

In the above picture, the root has 3 subtrees.

Each node can have arbitrary number of children. Nodes with no children called leaves, they also called external nodes. In the above picture, C, E, F, K, G are leaves. Nodes which are not leaves are called internal nodes.
Nodes with the same parent are called **siblings**. On the picture, B, C, D are siblings. The **depth of a node** is the number of edges from the root to the node. The depth of K is 2. The **height of a node** is the number of edges from the node to the deepest leaf. The height of B is 2. The **height of a tree** is a height of a root.

**Implementation**
Since each node in a tree can have an arbitrary number of children, and that number is not known in advance, the *general* tree can be implemented by using **first child/next sibling** method. Each node will have **TWO** pointers: one to the child, and one to the sibling. The following picture illustrates this

![Tree Diagram]

*Important conclusion:* It's pretty straightforward to see that we need to deal only with **binary** trees - a tree that can have no more than two children.

**Definition:** Recursively, a binary tree is either empty or consists of a root, a left subtree and a right subtree. Here is a prototype for a tree node:

A binary tree in which each node has exactly zero or two children is called **a full binary tree** - there are no degree 1 nodes.

**A complete binary tree** is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height h has between $2^h$ and $2^{(h+1)}-1$ nodes.
Here is an example:

![full tree and complete tree](image)

**Question 1.** What is the maximum height of a binary tree with N nodes?

The longest tree is a linked list (this is so-called a degenerate case). The height is \#nodes - 1. For example, the maximum height of a binary tree that contains 31 nodes is 30.

**Question 2.** What is the minimum height of a binary tree with N nodes?

The minimum height is provided by a complete tree. The maximum number of nodes in a complete binary tree is

\[
N = 2^{h+1} - 1
\]

Resolve this with respect to height

\[
h = \log_2 (N + 1) - 1
\]

**Question 3.** Consider a tree in which each node contains a maximum of 4 children (a quad tree). What is the minimum height of a quad tree that contains 21 nodes, what is the maximum height of a quad tree that contains 21 nodes?

The maximum height is 20.

The minimum height is 2 and it is provided by

\[
N = \frac{4^{(h+1)} - 1}{3}
\]

\[
h = \log_2 (3N + 1) / 2 - 1
\]

**Question 4.** What is the maximum number of external nodes (or leaves) for a binary tree with height H?

\[2^H\]

**Question 5.** What is the maximum number of internal nodes (or leaves) for a binary tree with height H?

\[2^H - 1\]
Binary Search Trees

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The idea of a binary search tree is to store data so that it can be retrieved very efficiently.

A BST is a binary tree of nodes ordered in the following way:

1. each node contains one key (also unique)
2. the keys in the left subtree are < (less) the key in its parent node
3. the keys in the right subtree > (greater) the key in its parent node
4. duplicate node keys are not allowed.

Here is an example of a BST

Exercise

Draw the binary tree, which would be created by inserting the following numbers in the order given

50 30 25 75 82 28 63 70 4 43 74 35

BST provides log time access to each element. Consider an arbitrary BST of the height h. The total possible number of nodes is given by

$$2^{h+1} - 1$$

In order to find a particular node we need to perform one comparison on each level, or \((h+1)\) comparisons. Now, assume that we know the number of nodes n and we want to figure out the number of comparisons. We have to solve the following equation with respect to h:

$$2^h - 1 = n$$
we obtain

\[ h = \log \frac{n+1}{2} \]

This means, that a BST with \( n \) nodes has a maximum of \( \log(n) \) levels, and thus it takes at most \( \log(n) \) comparisons to find a particular node. This is the most important fact you need to know about BSTs.

Binary search trees work well for many applications (one of them is a dictionary or help browser) but they are limiting because of their bad worst-case performance \( O(N) \). A binary search tree with this worst-case structure is no more efficient than a regular linked list.

One of the approaches to balance a tree is to make the following restriction: at most one node has one child. This kind of tree is called a balanced BST. In a balanced tree at most one node has one child, all other nodes have two or zero children. Balanced search trees are trees whose height in the worst case is \( O(\log N) \). There exist over one hundred types of balanced search trees. Some of the more common types are: AVL trees, B-trees, and red-black trees. AVL trees were invented by two computer chess specialists in Russia, Adelson-Velskii and Landis in 1962 (hence the acronym AVL). AVL trees are identical to standard binary search trees except that for every node in an AVL tree, the height of the left and right subtrees can differ by at most 1.

See \( BST.java \) for implementation.

**Exercises**

a) find the minimum node;
b) compute the height of the tree;
c) compute the number of internal nodes;
d) compute the number of external nodes;
e) compute the number of leaves at a given depth.

**Question 6.** What is the complexity of building a BST with \( N \) nodes?
Traversing Trees

Consider the binary search tree

```
    38
   /   \
  5     45
 /   /   /
1   9   47
 /       /   /
8   15   46
/     /   /
13
```

There are several methods of traversing trees:

**InOrderTraversal** - left child, parent node, right child

In the above example, the in-order traversal is

1, 5, 8, 9, 13, 15, 38, 45, 46, 47

**PreOrderTraversal** - parent node, left child, right child

In the above example, the pre-order traversal is

38, 5, 1, 9, 8, 15, 13, 45, 47, 46

**PostOrderTraversal** - left child, right child, parent node

In the above example, the post-order traversal is

1, 8, 13, 15, 9, 5, 46, 47, 45, 38

This picture demonstrates the order of node visitation in postorder, preorder, and inorder traversals:
**LevelOrderTraversal** - processing nodes from top to bottom, left to right

In the above example, the level-order traversal is

```
38, 5, 45, 1, 9, 47, 8, 15, 46, 13
```

**Euler Tours:**

The three common traversal algorithms can be represented as a single algorithm by assuming that we visit each node three times. An Euler tour is a walk around the perimeter of a binary tree where each edge is a wall, which you cannot cross. In this walk each node will be visited (touched) either on the left, or in the below, or on the right. For a left these three visits happen one right after the other, whereas for interior nodes that is not a case. The Euler tour in which we visit nodes on the left produces a preorder traversal. When we visit nodes from the below, we get an inorder traversal. And when we visit nodes on the right, we get a postorder traversal.

**Exercise**

- Draw a binary tree T such that
  - each node stores a single character and
  - a preorder traversal of T yields BINARY and
  - a postorder traversal of T yields NRYAIB

```
    B
   / \
  I   A
 / \ / \
N   R Y
```

**Exercise**

- Draw a binary tree T such that
  - each node stores a single character and
  - a preorder traversal of T yields 6, 4, 2, 1, 3, 5, 10, 8, 7, 9 and
  - can inorder traversal of T yields 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

```
  6
 /  \
4   10
 /   /  \
2   5   8
 /   /   /  
1   3   7   9
```

**Exercise (*)** Implement a method that accepts two arrays – the first array holds nodes of the inorder traversal and the second holds nodes of the preorder traversal of the same binary tree – returns a correspondent tree.
See *BST.java* for recursive implementation of the above traversals.

**Complexity of traversal.**
Assume that $T(n)$ is the function that describes complexity of traversing a binary tree with $n$ nodes. This accumulates complexity for visiting the root – it takes constant time – and for visiting the left subtree $T(m)$ and the right subtree $T(k)$, where $m+k = n-1$. Therefore,

$$T(n) = c + T(m) + T(k)$$

We solve this recurrence equation asymptotically, by guessing that $T(n) < b \cdot n$. It follows that

$$T(n) = c + T(m) + T(k) < c + b \cdot m + b \cdot k = c + b \cdot (m+k) = c + b \cdot (n-1) = O(n)$$

**Exercise.** Describe a nonrecursive method for performing an inorder traversal of a binary tree.

**Exercise.** Implement a copy constructor

```
public BST(BST rhs)
```

that creates an exact copy of the *rhs* tree.

### Evaluating Arithmetic Expressions

Consider the following tree:

```
  +
  / \
+ 11
| / \
- * 
\ / \
/ * - 
/ / \
\ / \ 
+ 7 3 1 2
\ / \ 
2 * 
/ \ 
3 4
```

A vertex is labeled by an operation that is applied to its children. By convention, the minus sign has only a right child.

Inorder traversal evaluates the arithmetic expression;

$$-(2 + 3 \cdot 4)/7 + 3 \cdot 1 \cdot (-2) + 11$$

Preorder traversal gives the *prefix* notation;

```
++-/+2*3 4 7 **3 1-2 11
```

Postorder traversal gives the *postfix* notation;

```
2 3 4 * + 7 / - 3 1 * 2 - * + 11 +
```
Delete

Algorithm: (two steps)

- find a node (throw an exception otherwise)
- delete a node

We will consider three sub-cases for the deletion operation:

1. a node (to be deleted) has only no children
2. a node (to be deleted) has only left child
to delete this node, make its parent node point to its left child;
3. a node (to be deleted) has only right child
to delete this node, make its parent node point to its right child;
4. a node (to be deleted) has exactly two children
to delete this node, find the rightmost node in the left subtree and swap data between these two nodes. The rightmost node will be the node with the greatest value in the left subtree. Why do we need this node? One way of thinking about it is that you want to replace the node to be deleted with a node that has the biggest value in the left subtree. Alternatively, you can swap data between the node to be deleted and the leftmost node in the right subtree.

Note. A node to be actually deleted can have a left subtree.
See BST.java for implementation.

Consider the binary search tree

```
  38
   /    
  5     45
   /     /  
  1     9    47
   /     /   /  
  8     15   46
    /     
   13
```

and suppose we want to delete a node with the label 15. Here is a stack of recursive calls:

```
38.left = delete (5_node, 15)
5.right = delete (9_node, 15)
9.right = delete (15_node, 15)
delete (15_node, 15) returns 13_node
```

Because of the last two lines, the node 9 is connected to 13, and the node 15 is deleted (since there are no references pointed to it).

**Exercise.** If you delete a node from a BST and then insert it back, will you change the shape of the tree?
Traversing Trees using the Iterator class

Iterator is the API interface. A class, which implements the Iterator interface, should implement three methods

- boolean hasNext() - Returns true if the iteration has more elements.
- Object next() - Returns the next element in the iteration.
- void remove() - (optional operation) Removes from the underlying collection the last element returned by next().

We implement a preorder traversal by adding a new method iterator to the BST class. This method returns an iterator over the nodes of a binary tree in pre-order:

```
public Iterator iterator()
{
    return new PreOrderIterator();
}
```

We implement the PreOrderIterator class as an inner private class of the BST class. The PreOrderIterator class implements the Iterator interface. We use the Stack as an intermediate storage:

```
private class PreOrderIterator implements Iterator
{
    private Stack stk = new Stack();

    // Construct the iterator.
    public PreOrderIterator()
    {
        stk.push(root);
    }

    public Object next()
    {
        BNode cur = (BNode) stk.peek();
        if(cur.left != null)
        {
            stk.push(cur.left);
        }
        else
        {
            //if a left child is null
            BNode tmp = (BNode) stk.pop();
            while(tmp.right == null)
            {
                if (stk.isEmpty()) return cur;
                tmp = (BNode) stk.pop();
            }
            stk.push(tmp.right);
        }
    }
}```
Algorithm - If there is a left child, we push the child on a stack and return a parent node. If there is no left child, we check for a right child. If there is a right child, we push the right child on a stack and return a parent node. If there is no right child, we move back up the tree (while-loop) until we find a node with a right child.

Here is a stack after returning 38 and 5:

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;--- tmp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|5 |       |5 |<--- tmp |9 |<--- tmp
|38|       |38|       |38|
    |       |   |       |   |
```

We shall leave the method `remove()` unimplemented. Since a binary tree has a nonlinear structure, removing a node might cause the major tree rearrangement, which will lead to incorrect output from `next()`:

```
public void remove()
{
    throw new java.lang.UnsupportedOperationException();
}
```

**Game Tree**

We will be playing a two-person game in which the two players alternate making moves. The game can be represented as a tree where the nodes represent the current status of the game and the edges represent the moves. The game tree consists of all possible moves for the current players starting at the root and all possible moves for the next player as the children of these nodes, and so forth. The leaves of the game tree represent terminal positions (a win, a loss, a draw, a payoff). Each terminal position has a score. We will associate 1 with a win, 0 with a draw and -1 with a loss.

**Game of NIM**

There are several piles of sticks. A player may remove any number of sticks from only one pile. The player who takes the last stick loses. We represent the piles by a monotone sequence of integers, such as (1,2,3) or (2,2,5,9). Thus, (1,2,3) would become (1,1,3) if the player removed 1 stick from the second pile. Here is a game tree for the NIM game (1, 2, 2)

Suppose you are to make the first move; you can take either one of two sticks. The result of this is {1,2} or {2,2} or {1,1,2}. Then the opponent moves one or two sticks, and so on.
To analyze this game we start with assigning 0 or 1 to all leaves. We assign 1 to all leaves of level 3 – it’s a winning outcome. And we assign 0 to all leaves of levels 2 and 4 – that is a loss for you.

We will compute the scores of the internal nodes from the bottom up. If the parent node represents the opponent’s move, we choose the minimum score. If the parent node represents your move, we choose the maximum score.

To find the winning strategy you should follow “1”s, starting from the root. Once you made a mistake, the opponent should follow “0”s.