CSC 212 Programming Assignment # 3
Implementing and Using Binary Trees to Solve the Staircase Puzzle.
Due date: 14/11/2018

Guidelines:	This is an **individual** assignment.
	The assignment must be submitted to **Web-CAT**

Trees can be used to solve puzzles by exploring all possible sequences of actions. In this assignment, your goal is to solve the staircase puzzle, in which a person stands at the bottom of a staircase containing n steps, numbered from 1 to n as shown in Figure 1. This person can go up the stairs by moving either k_1 or k_2 steps at a time (for example, the person can move 1 or 2 steps at a time). The question we want to answer is: What are the different ways to climb the stairs?

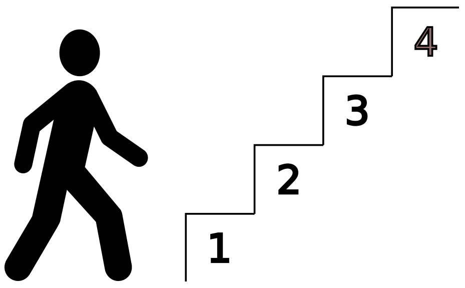

Figure 1: Example of the staircase puzzle. What are the possible ways to climb the stairs if you can move one or two steps at a time? In this example, $n = 4, k_1 = 1, k_2 = 2$.

To answer this question, we can build a binary tree that encodes all possible ways to climb the stairs. The root of this tree is the initial position of the person which is 0 (bottom of the staircase). From the initial position, there are two options (and hence two children), the person can either move one step to reach step 1 or move two steps to reach step 2. This process is repeated until the person reaches the top of the stairs or cannot move any more. For instance, the resulting tree for the example of Figure 1 is show in Figure 2. There are 5 ways to climb the stairs. Figure 3 shows the tree for the case: $n = 4, k_1 = 2, k_2 = 3$, and in this case there is only one way to climb the stairs.

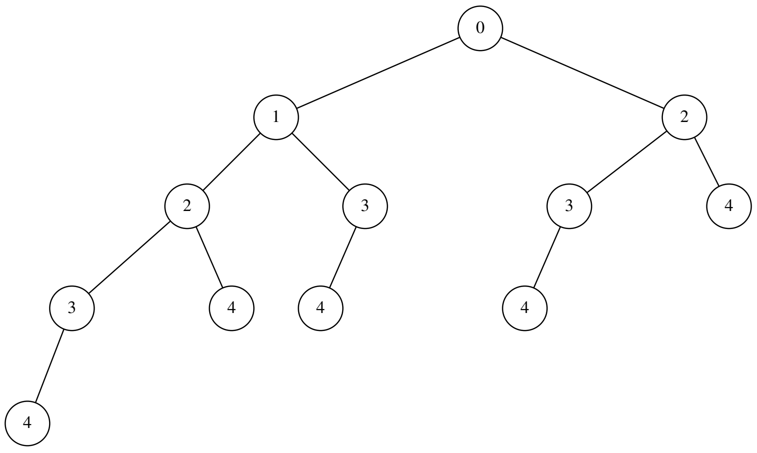

Figure 2: A binary tree encoding all possible ways to climb a staircase of four steps by moving either one or two steps at a time. There are five ways in total.

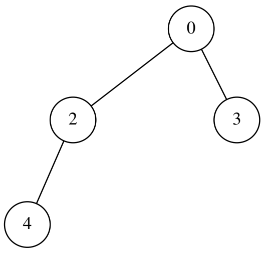

Figure 3: A binary tree encoding all possible ways to climb a staircase of four steps by moving either two or three steps at a time. There is only one way to do this: move two steps then again two steps. Notice that by moving three steps from the bottom of the staircase, the person gets stuck in step 3.

A variant of this problem consists in having one step that is broken, and hence the person cannot step into it as shown in Figure 4 and Figure 5.

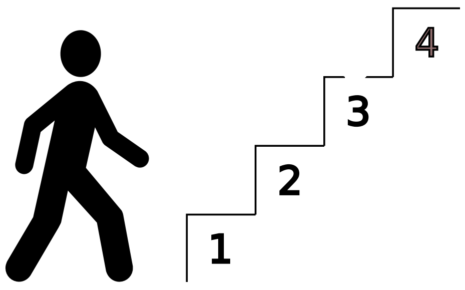

Figure 4: Example of the staircase puzzle with a broken step (step 3). What are the possible ways to climb the stairs if you can move one or two steps at a time, but without moving to step 3.

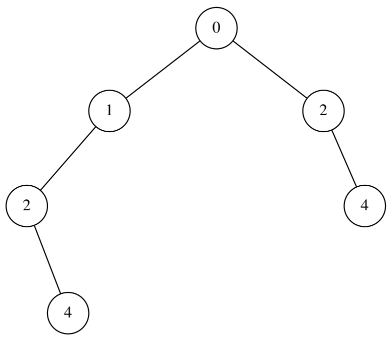

Figure 5: A binary tree encoding all possible ways to climb a staircase of four steps by moving either one or two steps at a time but without stepping into step 3.

To write a code that solves the staircase problem, we proceed in three steps:

1. Implement a binary tree data structure.
2. Write a class containing a set of user methods that will be used to solve the problem.
3. Write a class that uses the two previous classes to solve the staircase problem.

These three steps are detailed in what follows.

1 Implementing a binary tree

Given the following specification of the ADT Binary Tree, implement this data structure using linked representation. You should write the class `LinkedBT` that implements the interface `BT`. The class `LinkedBT` must represent nodes using the class `BTNode` shown below, which in addition to the left and right pointers, includes a pointer to the parent node as illustrated in Figure 6. All methods of the class `LinkedBT` must run in $O(1)$.

```
class BTNode<T> {
  public T data;
  public BTNode<T> left, right, parent;

  public BTNode(T data) {
    this.data = data;
    parent = left = right = null;
  }
}
```

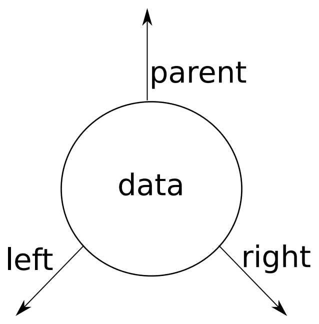

Figure 6: A binary tree node with a pointer to parent.

1.1 Methods seen in lecture

- full (boolean flag): **requires**: none. **input**: none. **results**: if the binary tree is full then flag is set to true otherwise it is set to false. **output**: flag.
- empty (boolean flag): **requires**: none. **input**: none. **results**: If Binary tree is empty then flag is set to true; otherwise flag is false. **output**: flag.
- retrieve (Type e): **requires**: Binary tree is not empty. **input**: none. **results**: element in the current node is copied into e. **output**: e.
- update (Type e): **requires**: Binary tree is not empty. **input**: e. **results**: the element in e is copied into the current node. **output**: none.
- insert (Type e, Relative rel, boolean inserted): **requires**: (1) full() is false, and (2) if rel = Root then empty() is true, otherwise, rel $\neq$ Parent and empty( ) is false. **input**: e, rel. **results**: if (rel = leftChild and current node has a left child) or (rel = rightChild and current node has a right child), then inserted is false. Otherwise a node containing e is added as rel of the current node in the tree, the new node becomes the current node, and inserted is true. **output**: inserted.
- find (Relative rel, boolean found): **requires**: Binary tree is not empty. **input**: rel. **results**: The current node of the tree is determined by the value of rel and previous current node. **output**: found.

1.2 New and updated methods

- deleteSub(): **requires**: Binary tree is not empty. **input**: none. **results**: The subtree whose root node was the current node is deleted from the tree. If the resulting tree is not empty, then the **parent** of current becomes the current. **output**: none.
- isLeaf (boolean flag): **requires**: Binary tree is not empty. **input**: None. **results**: if the current node of the binary tree is a leaf then flag is set to true otherwise it is set to false. **output**: flag.
- getRoot(`BTNode<T>` t): **requires**: none. **input**: none. **results**: t is set to the root of the tree. **output**: t.

  Remark 1. *This method clearly violates the principle of encapsulation and is only added for testing purposes.* ***You must not use this method in your code.***

2 User methods

Write a class called `BTUtils`. In this class, you should implement the following static methods (**do not use** `getRoot(BTNode<T> t)`):

1. **public static** `<T>` **int** `nbLeaf(BT<T> bt, T e)`: This method counts and returns the number of leaf nodes that contain the data e. You should use the `find`, `isLeaf` and maybe other `BT` methods for counting.
2. **public static** `<T>` **void** `pruneBranch(BT<T> bt, T e)`: This method prunes (cuts) all subtrees of the tree `bt` with root `e`. You should use the `deleteSub`, `find` and other `BT` methods as required.

3 Solving the staircase problem

Using the binary tree implementation and user methods above, write the class `Staircase` with the following static methods (**do not use** `getRoot(BTNode<T> t)`):

1. **public static** `BT<Integer> getTree(`**int** `n,` **int** `k1,` **int** `k2)`: Returns a binary tree of all possible actions for the staircase problem with n stairs, and k_1 and k_2 moves. Assume that $0 < k_1 < k_2$. The tree must be exactly as shown in Figure 2 and 3.
2. **public static** `BT<Integer> getTreeWithout(`**int** `n,` **int** `k1,` **int** `k2,` **int** `k)`: Returns a binary tree of all possible actions for the staircase problem with n stairs, and k_1 and k_2 moves, and broken step k. Assume that $0 < k_1 < k_2$. The tree must be exactly as shown in Figure 5 (obtained from the complete tree by pruning all paths passing through step k).
3. **public static int** `getNbSol(`**int** `n,` **int** `k1,` **int** `k2)`: Returns the number of solutions for the staircase problem with n stairs, and k_1 and k_2 moves. Assume that $0 < k_1 < k_2$.
4. **public static int** `getNbSolWithout(`**int** `n,` **int** `k1,` **int** `k2,` **int** `k)`: Returns the number of solutions for the staircase problem with n stairs, and k_1 and k_2 moves, and broken step k. Assume that $0 < k_1 < k_2$.

4 Deliverable and rules

You must deliver:

1. Source code submission to Web-CAT. You have to upload the following classed in a zipped file:

   - `LinkedBT.java`
   - `BTUtils.java`
   - `StairCase.java`

   Notice that you should **not upload** the interface `BT`, the enumeration `Relative` and the class `BTNode`.

The submission **deadline** is: **14/11/2018**.

You have to read and follow the following rules:

1. The specification given in the assignment (**class and interface names, and method signatures**) must not be modified. Any change to the specification results in compilation errors and consequently the mark zero.
2. All data structures used in this assignment **must be implemented** by the student. The use of Java collections or any other data structures library is strictly forbidden.
3. This assignment is an individual assignment. Sharing code with other students will result in harsh penalties.
4. Posting the code of the assignment or a link to it on public servers, social platforms or any communication media including but not limited to Facebook, Twitter or WhatsApp will result in disciplinary measures against any involved parties.
5. The submitted software will be evaluated automatically using Web-Cat.

6. All submitted code will be automatically checked for similarity, and if plagiarism is confirmed penalties will apply.
7. You may be selected for discussing your code with an examiner at the discretion of the teaching team. If the examiner concludes plagiarism has taken place, penalties will apply.