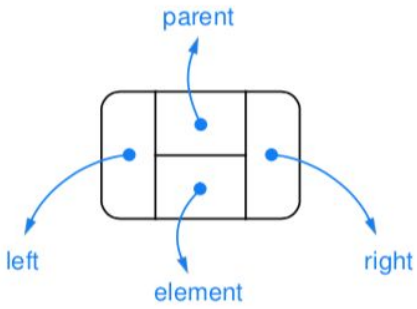
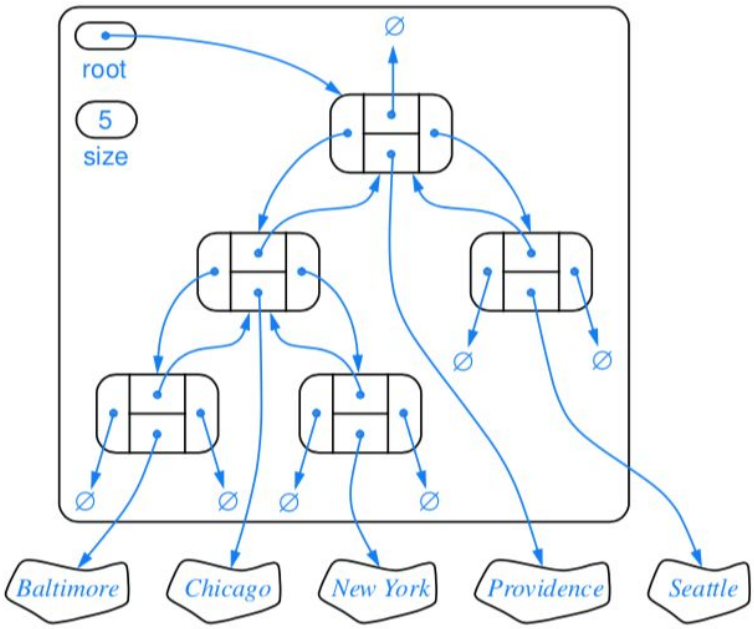
ACS-2947-050
Assignment #3
Due by Saturday November 17 at 11:59 pm

- Submit your `.java` files to 2947L-070@acs.uwinnipeg.ca or 2947L-071@acs.uwinnipeg.ca
- Include your name and student number in each file as a comment
    - Include comments as needed
    - Use appropriate exception handling where necessary (See note for Part A)

PART A

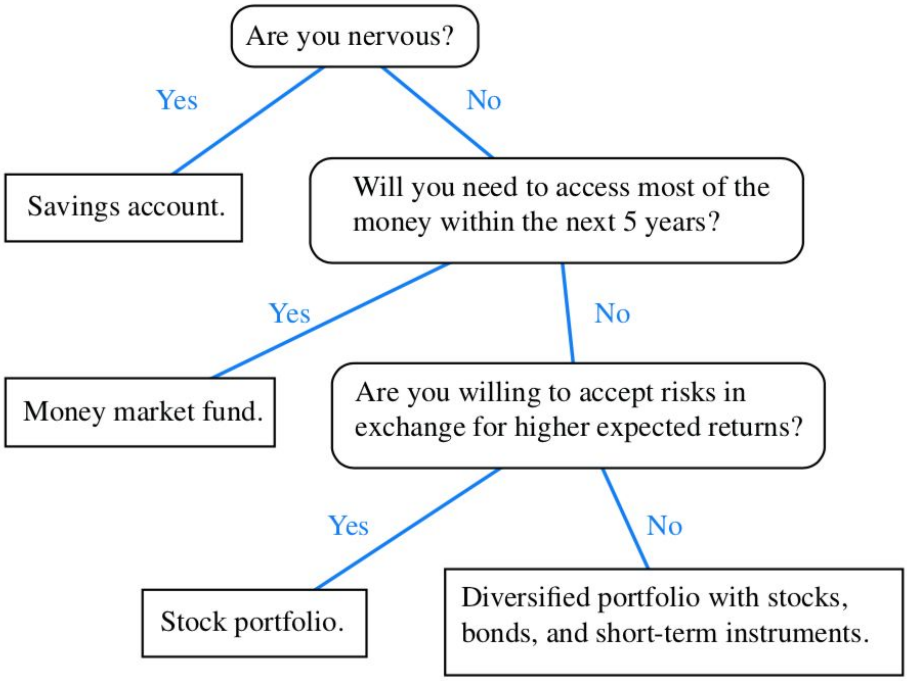
Implement the Binary Tree ADT using the ***generic linked structure*** shown in the following diagram:





1. Create a class called `LinkedBinaryTree` that implements the `BinaryTree` (which extends `Tree`) and `Position` interfaces. Use the `AbstractBinaryTree` and `AbstractTree` classes in your notes/textbook to provide a base for your linked binary tree.
2. Create an interactive program that asks the user to work through a Yes/No decision tree with height of at least 3.

    E.g.



Sample output:

```
System:     Are you nervous? (yes/no)
User:       no
System:     Will you need to access most of the money within the
            next 5 years? (yes/no)
User:       no
System:     Are you willing to accept risks in exchange for higher
            expected returns? (yes/no)
User:       yes
System:     Stock portfolio
```

Notes:

- Iterators are not required for this question but will be needed for Part B
  - Use placeholder code if not implementing immediately
- Build the tree by assigning/re-assigning positions as you go along.
- Map your yes/no to left/right child, and work through the decisions until an answer is reached (external node)
- Your code should work for any linked binary decision tree i.e. don't hardcode questions/answers to work only with your tree

PART B

Implement the Priority Queue ADT using a heap. The heap will use your ***LinkedBinaryTree*** from PART A and a comparator.

1. Create a class called `LinkedHeapPriorityQueue` that implements the `PriorityQueue` and `Entry` interfaces. Use the `AbstractPriorityQueue` class in your textbook to provide a base for your Linked Priority Queue.
2. Create a driver program to show you working with your priority queue in a simple simulation of a ***real-world*** example (e.g. airline standby list).

Notes:

Before starting Part B, make sure that your `LinkedBinaryTree` is fully implemented. You should have your tree traversal algorithms set and `toString()` in place.

First, have a good understanding of the array-based `HeapPriorityQueue` from your notes/textbook. Here, the parameters are indices that represent the level number of each entry. With a linked tree-based PQ the parameters will be `Position` objects. Instead of using indices to access entries in the tree, we will determine the positions of these elements relatively.

Start building your LHPQ. Declare a `LinkedBinaryTree` called `heap` that holds `Entry` objects as its elements. Add the constructors in the same manner as your textbook `HeapPriorityQueue`, and make sure that a `DefaultComparator` is included in your package. The next 5 protected utilities of HPQ are not required in the LinkedHeap version because all of this information can be either directly accessed or quickly determined via the LBT methods.

Next, look at the protected `swap` utility: instead of indices (int), you will have `Position` objects as parameters. In an array, you swap the elements in the given array indices. How would you swap the elements in given positions? Use this to form a basis of how to convert from array-based to LBT-based.

Suggestions:

- Override the `toString()` method to help with debugging
    - Should be quick if the `toString()` in your LBT is in place
    - Using the breadth-first traversal algorithm can be handy with PQs.

- The first method that you need to get running is `insert` (which needs `upheap` and `size` in place): use simple sample data as you are building/testing i.e. k-v pairs 8-8, 6-6, 7-7, 5-5, 3-3, 0-0, 9-9
    - This way you will insert entries that may or may not need upheaping, and values outputted are easier to understand and map
    - Jot down what the heap should look like and compare when debugging
- You will need to find a way to insert the next node to satisfy the complete binary tree property: think of how a *binary* tree works:
    - How can we insert a new entry in the next position? I.e. how do we find the parent to add this new position to, and whether we add to left or right?
    - We will discuss this algorithm next class

- Once your `LinkedHeapPriorityQueue` is in place and working accurately then you can think of a real-world application to simulate.