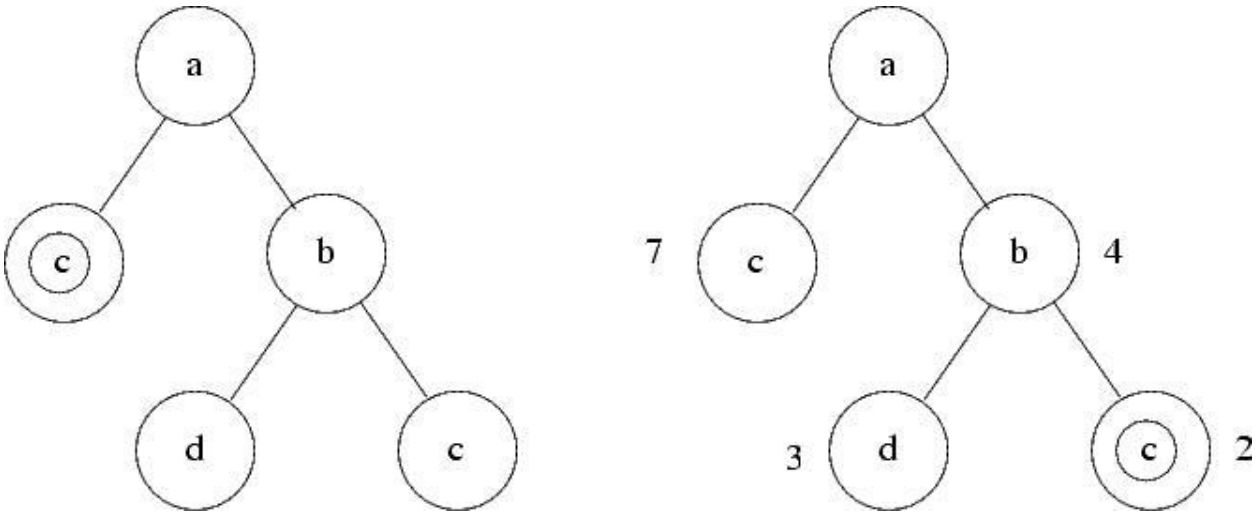
Data Structures Assignment - Modified BFS

We have seen that when searching for an item in a tree, BFS will return the sought for item that is closest to root. In this scenario we can imagine that the vertices in the tree are equally far from each other. In other words, the distance from the root to a vertex is just the level of that vertex, or alternatively the number of edges traversed to get from the root to the vertex. In this assignment, we want to generalize our distance measure to allow for arbitrary distances - each edge defines a distance between the two vertices it connects.



Here we see that in the left tree if we're searching for letter 'c', BFS will return the highest level 'c' in the tree. In the case of variable distances, we want our search to return the 'c' that has the shortest distance, where the distance of a node *v* is calculated by summing the numbers along the path from root to *v*.

In this case, using a regular queue will not do, because the vertex that is the "fewest hops away" may not be the closest. Instead we need a queue that takes the vertex distances into account, and give priority to the closest ones. This priority queue is exactly the min-heap we've already used many times! So, as DFS uses a stack and BFS uses a queue, this algorithm (a simplified version of Dijkstra's algorithm) uses a min-heap to order the vertices it explores.

Assignment #7:

1.) Supporting variable distances
First, you should modify the binary tree class so that it can store distances. This can be done simply by adding an extra field in BinaryNode (and changing the name to TreeNode):

```
public class TreeNode {
  //...
  double distance;
  //...
}
```

This field should initially hold the distance the node is from its parent. You will also want to make sure TreeNode implements the Comparable interface, this means adding a compareTo method. The compareTo should of course use the distance values to decide its answer. Finally, you should make TreeNode a generic class, so the class definition should actually read:

```
public class TreeNode< AnyType > {
     AnyType element;
     //...
}
```

Where in our examples, we will be using TreeNode< String >, since our node labels will be letters "a", "b", etc.

2.) Optimal Distance Search
Next, we need to modify the existing BFS search to take advantage of a min-heap. Here is the pseudo-code:

```
findClosest(Node root, Object target) {
make new min-heap and insert root with distance 0.
 while min-heap is not empty {
    q <- get min element
    while q not null, and q does not hold target:
      place q's children into heap, using distance from root as the
sort key.
  }
```

Notice that when we insert a child into the heap, we need to make sure its "distance" field (which we sort by) is updated to be its distance from the root rather than its parent. You can easily calculate its distance from the root if you simply set child.distance += parent.distance, each time you place a child into the heap. Obviously, you don't need to do this for the root node since it has no parent (just set its distance to 0 initially).

3.) Reading Input
For simplicity's sake, you may assume that the input to the algorithm encodes a binary tree, where every node has either zero or two children. The input will be provided in a text file (whose name will be provided as a command-line argument) and will be formatted as follows:
a 0 (b 3 c 7 (d 1 e 3))

This encodes the tree with node 'a' set as the root, with children 'b' that is 3 units away and 'c' which is 7 units away. Furthermore, node 'c' has its own children 'd' that is 1 unit away from 'c' (not the root) and 'e' which is 3 units away. Every token will be separated by whitespace, so you can easily use the Scanner class to read the input token by token.

To read the input we will need to use a stack and the following algorithm:

```
While hasNextToken {
  temp <- getNextToken
  if temp equals "("
    do nothing
  if temp equals ")"
    pop off last two nodes (call them left_child, right_child)
    pop off a third node (call it parent)
    assign left_child, right_child as parent's children
    push parent back onto the stack
  else //we read in a node label
    push a new TreeNode(label=temp, distance=getNextToken)
}
```

It should be clear that once this algorithm has completed, there should be a single TreeNode remaining on the stack. This tree node will be the root of the tree described by the input. If this is not clear you should work out a couple examples on paper. Also, notice that TreeNode will need a function that allows you to set children (or allow public access to its child pointers).

Bonus:
For those of you wanting an extra challenge. Notice that the "(" in our input didn't serve any purpose. That is because we fixed the number of children to exactly two. We could also allow each node to have an arbitrary number of children. Each node would then hold a linked list of children, and the input could look like the following:
a 0 (b 3 c 8 (f 3 g 2) d 3 (e 3))

Where 'a' has 3 children, 'b' has 0 children, 'c' has 2 children and 'd' has one child.

4.) Examples:
Your program should read a tree input and search for the special string "*", it should then return the distance at which it was found or null if it was not found. A correct algorithm will return the shortest such distance if "*" appears multiple times. So given the input:
a 0 (b 4 (* 100 b 6) w 9 (x 3 y 5 (* 2 z 3)))

We should expect an output:
Found "*" at distance 16.