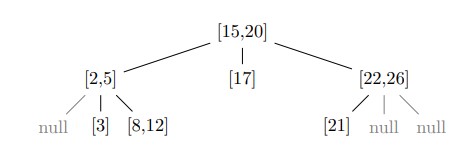
**1)Wide Search Tree**

In this section, you will implement wide search trees of integers, in a class called WideSearchTree. A wide search tree is like a binary search tree but nodes of the tree may store more than one data item and have more than two children. Specifically, a search tree of width w has nodes that stores up to w data items and have up to w + 1 children. A node can only have children if it contains exactly w data items. For simplicity, wide search trees cannot contain duplicates.

 Binary search trees are width-1 search trees. Each node stores one integer and has up to two children. Recall that the numbers in a node’s first (left) subtree must be less than the integer stored in the node, and the numbers in the second (right) subtree must be greater. In a width-2 search tree, each node stores up to two integers and has up to three children. If there are two integers, call them a and b. They are stored in order, so let’s say that a < b. The first subtree contains integers less than a, the second subtree contains integers between a + 1 and b − 1 inclusive, and the third subtree contains integers greater than b. Here is an example:

The grey nulls indicate that the parent node has no child in that position (i.e., the node storing 2 and 5 has no first child, and the node storing 22 and 26 has no second or third child). I’ve not shown the nulls for nodes that have no children (i.e., where all three children are null). Notice that the root’s first (left) subtree contains only values less than 15, its second (middle) subtree contains only values between 16 and 19, and its third (right) subtree contains only values greater than 20.

If we insert 1 into the above tree, it will be placed as the first child of [2, 5] in a new node that replaces the null. The root node is full, and 1 < 15 so we go left; [2, 5] is full, and 1 < 2 so we add a first child. If we insert 4, it will be stored in the node that currently contains [3]. This is because 4 is between 2 and 5, and the node [3] is not full, so we add 4 there. Trees with width w = 3, 4, 5, . . . are defined analogously. Nodes contain sorted lists of up to w numbers. If there are w numbers in a node, it may have children. If the numbers stored in a node are n1, . . . , , sorted in increasing order, then the first child contains numbers less than n1, the last child contains numbers greater than and the ***i*** th child contains numbers between and − 1. Nodes that store fewer than w numbers cannot have children.

* 1. **Setting up the tree (6%)**

Create a class called *WideSearchTree* to represent wide search trees of integers. The class should have a constructor that takes an *int* argument specifying the width, and it should create an empty tree.

There should be an inner class called *Node* that represents a single node of the tree. You may use any appropriate way of storing the data values and the subtrees, such as arrays or lists. If you’re using lists, use one of Java’s standard list classes – there’s no need to write your own.

* 1. **Making an example (6%)**

Write a method *static WideSearchTree getExample()* that creates the example tree above, by “manually” creating *Node* objects and linking them together in the right way.

* 1. **Searching (7%)**

Write a method *boolean contains(int x*) that determines whether the tree contains the value *x*. The method is similar to searching binary search trees, but with more cases to consider.

* 1. **In-order traversal (7%)**

Write a method *void printInOrder()* that uses an in-order traversal to print out the values in the tree. Again, the method is a generalization of in-order traversal for BSTs

* 1. **Insertion (10%)**

Write a method void insert(int x) that inserts the value x into the tree. The algorithm is as follows:

1. If the tree is empty, just create a node holding value x, make it the root and return. 2.
2. Otherwise, start at the root. 3.
3. If the current node has fewer than w values, store x in the current node and return.
4. Otherwise, move to the appropriate child and go to step 3; if the child does not exist, create it.

Where appropriate, check that x is not already in the tree. Remember that you only create children of nodes whose own list/array is full. If the list/array isn’t full, any new data is stored in the node itself.

* 1. **Height (7%)**

Write a method *int height()* that returns the height of the tree. Use the convention that the empty tree has height −1 and a tree with just a root has height zero.

* 1. **Testing (7%)**

Write a main method that performs the following tests.

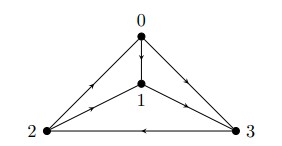
1. Create the example tree. Print its contents, check that it has height 2, check that it contains 3, 5, 20 and 21 and does not contain 1, 4 and 13. Insert 13 and check that the height is now 3.
2. Create a tree of width 4 and insert the numbers 0–999 in a random order. You can do this by creating a list of the numbers, using *Collections.shuffle()* and then inserting the elements from the list into the tree. Check that the tree contains every number from 0 to 999 inclusive, and that it does not contain −1 or 1 000. Calculate the average height of 1 000 trees, each made from a random shuffle. The answer should be about 7.0–7.1.

**2) Important nodes in graph**

This section is about measuring the importance of vertices in a directed graph. The idea is that important vertices have lots of incoming edges and, especially, lots of incoming edges from other important vertices. Vertex importances are fractional numerical values so should be stored in doubles. The importance of vertices is calculated by an iterative algorithm. Initially, each vertex is given importance 1.0. Then, the algorithm iterates. At each iteration, each vertex shares its importance equally among its out-neighbours. We keep iterating until the importance values stop changing – it turns out that this is guaranteed to happen. As the process runs, vertices with high in-degree accumulate a lot of importance from their neighbours. To do this, we initially assign importance 1.0 to every vertex. We then iterate, setting the importance of vertex v at the current iteration to be

Where is the set of v’s in-neighbours, imp(w) is w’s importance at the previous iteration, and is w’s out-degree.

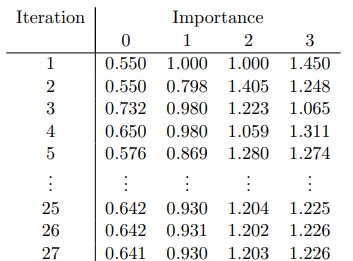
For example, consider the graph



Initially, we set every vertex’s importance to 1.0

At the first iteration, we set 0’s importance to 0.1+0.9×imp(2)/ = 0.1+0.9×1.0/2 = 0.55. 1’s importance becomes 0.1+0.9×(imp(0)/ + imp(2)/ = 0.1+0.9×(1.0/2+ 1.0/2) = 1.0. Notice two things, here: first, 1’s importance didn’t change; second, we used 0’s importance of 1.0 from the previous iteration, not the value 0.55 that we calculated just now. 2’s new importance is 0.1 + 0.9 × imp(3)/ = = 0.1 + 0.9 × 1.0/1 = 1.0 and 3’s importance becomes 0.1 + 0.9 × (imp(0)/ + imp(1)/ = 0.1 + 0.9 × (1.0/2 + 1.0/1) = 1.45 At the second iteration, 0’s importance becomes 0.1 + 0.9 × 0.1/1 = 0.55, 1’s becomes 0.1 + 0.9 × (0.55/2 + 1.0/2) = 0.798 and so on.

We get the following values:



You can see that the values take a little while to settle down. However, they change by less than 0.001 from iteration 26 to 27, so we stop calculating there. (The values in the table are rounded, so some of them appear to change by exactly 0.001 between those iterations.)

**2.1 Creating an example (8%)**

Create a class called ***VertexImportance***. Write a *method static Graph* ***getExample()*** that creates the example graph above and returns it. Use the ***MatrixGraph*** class to represent the graph.

**2.2 Augmenting the graph (8%)**

Write a method ***static void augment(Graph g)****.* For every vertex *v* in the graph g that has no outgoing edges, the method should add edges from v to every other vertex.

Note 1. This probably seems like a strange and arbitrary thing to do. Having vertices of out-degree zero causes problems because importance flows into them and then disappears because it doesn’t flow out again. Adding these edges solves this problem and it actually makes sense in the mathematical model that underlies the equations.

Note 2. The example graph from the previous part doesn’t have any vertices with outdegree zero, so running augment() on it shouldn’t change it.

**2.3 Computing vertex importance (15%)**

Write a method ***static double[] getVertexImportance(Graph g)***, which calculates the importance of each vertex in g and returns the answer as an array of doubles, such that the i th entry of the array is the importance of vertex i. The algorithm is as follows.

1. Augment the graph.

2. Initialize the array to all-1s.

3. For each vertex v, calculate its new importance using the formula above.

4. Repeat step 3 until no vertex’s importance changes by more than 0.001.

**2.4 Random graphs(12%)**

Write a method *static Graph* ***getRandomGraph()*** that creates and returns a random directed graph as follows. There are 20 vertices. We will refer to vertices 0–4 as level 0 and vertices 5–9, 10–14 and 15–19 as levels 1, 2 and 3, respectively. For each vertex x in level 0 and each y in level 1, add the edge (x, y) with probability 1/3. Do the same between levels 1 and 2 and between levels 2 and 3

**2.5 Testing(7%)**

1. Generate the example graph and calculate and print the vertex importances. You should get close to the answers that I got, though you might get slightly different answers due to different rounding from slightly different methods.

2. Generate 1, 000 of the random graphs. Compute the vertex importances and calculate the average of vertices 0–4 across all the graphs. The average should be about 0.48