CS 1501 Algorithm Implementation
Summer 2020
Programming Project 2

Online: Thursday, May 28, 2020
Due: All assignment materials: 1) All files necessary for your trie implementation, correctly stored within the TriePackage directory. These files include: TrieNodeInt.java (provided), MTAlphaNode.java, DLBNode.java and HybridTrieST.java, 2) The HybridTrieTest.java main program file (provided) 3) Any other files that you have written and 4) Assignment Information Sheet. All materials must be zipped into a single .zip file and submitted **via the course submission site** by **11:59 PM on Friday, June 12, 2020**.
Late Due Date: 11:59PM on Monday, June 15, 2020

Note 1: Do not submit a NetBeans or Eclipse (or any other IDE) project file. If you use one of these IDEs make sure you extract and test your Java files WITHOUT the IDE before submitting.

Note 2: Do not submit your revdict.txt or output dictionary files – they are too large. The TA will have a copy of the revdict.txt file to use for testing.

Background:
Now that we have discussed implementing tries using arrays for the nodes and also linked-lists for the nodes (DLBs) we would like to implement a hybrid trie which attempts to benefit from both implementations. As discussed in lecture, the DLB implementation is clearly more memory efficient in nodes with few children, with a relatively small time cost for traversing the list of nodelets in order to find the appropriate child at each node. However, as the number of children grows, the DLB implementation begins to suffer in two ways: 1) The memory can actually surpass that required of the array-based nodes, since each DLB nodelet requires more memory than a single pointer in an array and 2) The time to find a child becomes significant as the list of children in a DLB node grows longer.

Thus, it makes sense to implement a trie with the following policy:

- Initially make a new node in the trie a DLB node
- Monitor the degree (number of children) of each node in the trie. If the degree gets past a certain threshold, convert the node to an array-based node and use that node for future access.

Even though a DLB node and an array-based trie node are very different in structure, since they will have the same functionality we can represent both as implementations of the TrieNodeInt<V> interface. We discussed the TrieNodeInt<V> interface in lecture and saw a simple implementation (MTNode.java). In this assignment we will be using an enhanced TrieNodeInt<V> interface to allow additional functionality in each node and thus in our trie overall.

Details:
Consider the interface TrieNodeInt<V> in file TrieNodeInt.java. **[Important Note: Be sure to get the TrieNodeInt.java file from the Assignments page and NOT the one from the Handouts page. The version on the Assignments page is newer and has more methods]**.

This interface defines the functionality of a single trie node. As discussed in lecture, we can use this interface as the basis for our overall trie structure. **Read this interface very carefully, and note the comments describing each of the methods.** Many of the methods should be familiar to you, but there are several new methods that you should also note.

You must create two implementations of the TrieNodeInt<V> interface:
`MTAlphaNode<V>` (in file MTAlphaNode.java):
This implementation is similar to the one provided for you in the handout from lecture (MTNode.java). However, for this implementation we will limit our nodes to lower case letters, so there will be only 26 children per node. Also, since all character values are no longer valid indices in the array, you will need to map the character values to the correct positions in the array, throwing an exception if the character is invalid. In addition, there are new methods to consider in the interface – think carefully about how you will implement these.

Finally, in order to allow your DLBNode<V> nodes to be converted to MTAlphaNode<V> nodes as described above, you will need a constructor for your MTAlphaNode<V> class that takes an argument of a DLBNode<V>:

```
public MTAlphaNode(DLBNode<V> oldNode) {}
```

This constructor will need to iterate through the nodelets in the DLBNode<V> and assign the appropriate child pointers to the MTAlphaNode<V>. To do this, your MTAlphaNode<V> class will need access to the instance variables within the DLBNode<V> class. Thus, you should make your DLBNode<V> instance variables **protected** rather than **private** (since being in the same package allows access to protected variables).

`DLBNode<V>` (in file DLBNode.java)**:**
This implementation will be of a de la Briandais tree node as discussed in lecture. The children of each DLBNode<V> will be represented in a linked-list of nodelets. Thus, you will need an inner class in your DLBNode<V> class to represent a nodelet (child reference, sibling reference, character). See course notes for some details on the DLBNode<V> structure. To allow for access of your inner class and instance variables by the MTAlphaNode<V> constructor, make all of these declarations protected rather than private.

Clearly the TrieNodeInt<V> interface methods will be implemented very differently in this class, as discussed in lecture. For example, the method getNextNode(char c) in the MTAlphaNode<V> class simply returns the reference at the index corresponding to c. However, in the DLBNode<V> class this same method will require a traversal of the linked-list of nodelets until either the character is found or the end of the list is reached. Think about the DLBNode<V> structure when considering the implementations of the other interface methods.

Note that in the setNextNode(char c, TrieNodeInt<V> node) method a new nodelet may be created to store the reference to the argument node. For the trie functionality this nodelet could be anywhere within the linked list corresponding to the current DLBNode<V>. However, since we want to keep the strings in **alphabetical order**, your setNextNode() method should make sure that the new nodelet is placed into the proper position within the linked list based on the character values. In other words, if, within a trie node, we have children corresponding to 'a', 'b' and 'c', the nodelets should be in that order within the linked list.

To utilize your TrieNodeInt<V> implementations, you will also write a new trie class:

`HybridTrieST<V>` (in file HybridTrieST.java):

You will have **two instance variables** in your HybridTrieST:
The first is a reference to the root of your tree:

```
private TrieNodeInt<V> root;
```

Note that this reference is of type TrieNodeInt<V> and thus could refer to either an MTAlphaNode<V> or to a DLBNode<V>.

The second instance variable in your HybridTrieST is an int to indicate how the trie will create new nodes. This variable will have 3 possible values:

0: use MTAlphaNode<V> nodes for all nodes in the trie
1: use DLBNode<V> nodes for all nodes in the trie
2: use nodes of both types for the tree, as explained above

and it will be initialized in the constructor for the trie.

In the case of 2, the threshold for converting a node from a DLBNode<V> to an MTAlphaNode<V> is **when its degree gets to the value 11**. The justification for this is based on the memory required for each node. If we assume that each Java reference requires 4 bytes of memory and that a char requires 2 bytes of memory [1], and **considering only the memory required for branching**, we get the following:

A DLBNode<V> requires 10 bytes for each nodelet (child reference, sibling reference, char value), representing each child of the node

An MTAlphaNode<V> requires 26 x 4 = 104 bytes for all of its child pointers in its array, whether they are used or not

Thus, for degree 10, a DLBNode<V> requires 10 x 10 = 100 bytes for its child references, but for degree 11 it requires 11 x 10 = 110 bytes, which is more than the 104 bytes required for the MTAlphaNode<V>. Thus, from degree 11 and up, the MTAlphaNode<V> is more memory efficient than the DLBNode<V>.

Note that both the MTAlphaNode<V> and the DLBNode<V> will also require:

1) A reference to the array or front of the linked list. This will be 4 bytes for each implementation.
2) A variable to store the degree of the node. This will be an int and will be 4 bytes for each implementation.
3) A reference to store the value for the node. This will be 4 bytes for each implementation.

In order to convert properly from a DLBNode<V> to an MTAlphaNode<V>, you will need to test the actual type of the node in the trie as well as its degree. The degree is a method in the TrieNodeInt<V> interface, and the type can be tested using the **instanceof** operator in Java. Due to Java type restrictions, to test the types of these nodes, you should use the following syntax:

```
if (currentNode instanceof DLBNode<?>) …
```

Note that the parameter for the DLBNode is ? rather than a specific type value such as V. You will also need to test the type of a node for the countNodes() method.

Overall, you can see the requirements for the HybridTrieST<V> class in the test file HybridTrieTest.java. Read this file very carefully and note how the methods are called and what they are required to do (via the comments). Also see the expected execution in the test output files: Simple-out.txt, MT-out.txt, DLB-out.txt and Hybrid-out.txt. Simple-out.txt uses dictionary simple.txt and the other three use dictionary revdict.txt. **Your output should match the output shown in these test files exactly.**

To get you started with this project I have provided skeletons for the three classes that you must write. These contain the class headers and data declarations. You must provide all of the methods so that these classes work with the HybridTrieTest.java program. See:

DLBNode.java MTAlphaNode.java HybridTrieST.java

Important Notes:

– In your MTAlphaNode<V> class you will be making an array of a generic type. See handout MTNode.java from the CS 1501 handouts page for how to do this without generating a compilation error (however, it will generate a

warning – that is ok).

- Generally, you can get a lot of help with this assignment by carefully reviewing the course handouts: MTNode.java and TrieSTMT.java. Your MTAlphaNode<V> class should be very similar to MTNode<V>. Also your HybridTrieST<V> class will have a lot of similarities with TrieSTMT<V>. You could even use this class as a starting point for your HybridTrieST<V> class if you wish, but make sure you are aware of the differences in the two classes.
- You must utilize the Java Iterable interface in this project. This is a very useful interface and you should look it up in the documentation for more information. However, you don't need to know it in full detail to use it here – you just need to know the idea of it.

Extra Credit: If you add some non-trivial extra functionality to your HybridTrieST<V> you can get some extra credit. Think about what might be useful to do and make sure it is implemented using the TrieNodeInt<V> interface exclusively (i.e. independent of a particular node type).

---

[1] The size of a Java reference is not defined within the API, but implementations typically use 4 bytes as long as the heap is under 32Gb, which will be the case for us. A Java char requires two bytes, which is more than we need here, but for convenience we will use a char. A byte would save us some memory but would make our operations a bit more complicated.