EGR 227 — Data Structures and Analysis
Project #1 (HW#7) Zip
Checkpoint 1 (1)-(3) Due: 3/27 Fri 10 PM
Checkpoint 2 (4)-(6) Due: 4/3 Fri 7 PM
Final (7)-(8) Due: 4/13 Mon 7 PM

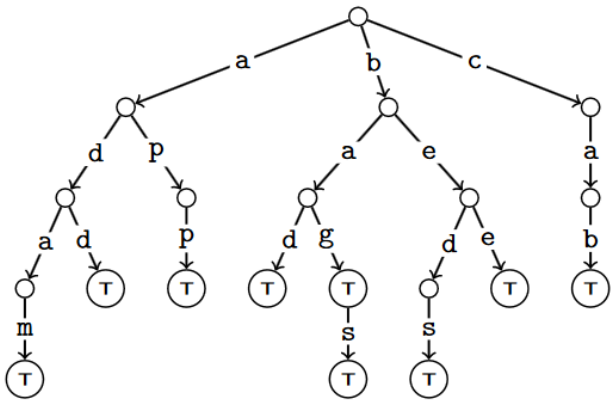
The purpose of this project is to 1) implement various "WorkList" data structures, 2) to learn an important new data structure, and 3) to implement a real-world application. **This is a team project of two sharing the same team repository.** Do not modify all files under egr221a.* package. The code you submit must work properly with the unmodified versions. All supplementary files are provided as part of the starter code.

Overview

A `WorkList` is a generalization of Stack, Queue, etc. A `WorkList` contains items to be processed in some order. The `Worklist ADT` is defined as follows:

`add(work)`	Notifies the worklist that it must handle **work**
`peek()`	Returns the next item to work on
`next()`	Removes and returns the next item to work on
`hasWork()`	Returns true if there's any work left and false otherwise

A `Trie` is a type of dictionary made for storing "words" (types made up of letters). You've actually already seen tries in HW6. We will describe them in full detail later, but for now, here's an example:



This trie represents the dictionary: `{adam, add, app, bad, bag, bags, beds, bee, cab}`, because if we go from the root of the trie reading in letters until we hit a "true" node, we get a word. Recall that in huffman, we had two possibilities (0 and 1) and we read from the root to a leaf.

In this project, you will implement several different types of `WorkLists` and a generic trie. Upon completing `SuffixTrie`, you will also be able to use these structures to compress inputs into a `*.zip` file which can interoperate with the standard `zip` programs!

Checkpoint 1: (1) – (3)

`ListFIFOQueue`, `ArrayStack`, and `CircularArrayFIFOQueue`. You must finish implementing all methods for these data structures. You will be given the opportunity to fix bugs and add comments and refactor your code to polish by Checkpoint 2. However, I strongly encourage to do your best to make them as complete as possible by Checkpoint 1. Only minor changes should be made on (1)-(3) after Checkpoint 1 and by Checkpoint 2. JUnit tests are provided for you to test the basics in `tst/TestCheckPoint1.java`, which are provided as part of the starter code. These tests are not exhaustive, thus you may write your own tests to ensure the correctness. In order to submit, push your to GitHub repository and upload a Word doc to BB with screenshots containing the root of your GitHub repository and showing the test results of JUnit tests for `TestCheckPoint1`.

Checkpoint 2: (4) – (6)

`MinFourHeap`, `HashTrieMap`, and `HashTrieSet`. You must finish implementing all methods for these data structures. You will be given the opportunity to fix bugs and add comments and polish your code by the final deadline. However, I strongly encourage to do your best to make them as complete as possible by Checkpoint 2. Only minor changes should be made on (4)-(6) after Checkpoint 2 and by the final deadline. JUnit tests are provided for you to test the basics in `tst/TestCheckPoint2.java`, whichare provided as part of the starter code. These tests are not exhaustive, thus you may write your own tests to ensure the correctness. In order to submit, push to GitHub repository and upload a Word doc with screenshots containing the root of your GitHub repository and showing the test results of JUnit tests for `TestCheckPoint1` and `TestCheckPoint2`.

Final: (7) – (8)

`WriteUp` and `SuffixTrie`. Manage your time well from Checkpoint 1 so that you have sufficient time left to finish the write up and the implementation of `SuffixTrie`. You must design good experiments for writing (7). JUnit tests are provided for you to test the basics in `tst/TestCheckPoint3.java`, which are provided as part of the starter code. These tests are not exhaustive, thus you may write your own tests to ensure the correctness. In order to submit, push to GitHub and upload a Word doc with screenshots containing the root of your GitHub repository and showing the test results of JUnit tests for `TestCheckPoint1`, `TestCheckPoint2`, and `TestCheckPoint3` - only if you have implemented (8).

Starter Code

Get the starter code from https://classroom.github.com/g/xz-3SA-Q

This is a team repository shared by both you and your teammate. Github will prompt you to create a new team or join an existing team. The first person who accepts the invite will create a new team. Feel free to pick any name that is appropriate ☺). The second person will join the team the first person created.

Note your team github repo should be something like

https://github.com/cbu-egr227-sp20/hw7-{YourTeamName}

Submission Instruction

For each Check Point, push to git and submit a Word doc to BB with the required screenshots.

File Sharing with Git

Please refer to BB for the "Basic Collaboration Scenario with Git" for details.

Project Restrictions

- You must work in a group of two unless you successfully petition to work by yourself.
- The design and architecture of your code are a substantial part of your grade.
- Your `WorkList` implementations may not use any classes in `java.util.*` (includes `List`, `Set`, `Map`, `Arrays` class, etc).
- Exceptions (like `java.util.NoSuchElementException`) are allowed.
- You may not edit any file in the `egr221a.*` packages.
- For better or worse, this project goes up against the limits of Java's implementation of generics. You will have to deal with this, but it is not a goal of this project for you to completely understand how these work. If you get stuck with generics, please ask for help immediately!
- Make sure not to duplicate fields that are in super-classes. This will lead to unexpected behavior and failures of tests!

Provided Code (Do not modify these files!)

- `egr221a.interfaces.misc`
  - `Dictionary.java`: An interface representing a generic Dictionary data structure.
  - `Set.java`: An interface representing a generic Set data structure.
  - `SimpleIterator.java`: An interface representing an iterator for a data structure.
- `egr221a.interfaces.worklists`
  - `WorkList.java`: An interface representing a generic WorkList
  - `FIFOWorkList.java`: An interface representing a WorkList that stores items in FIFO order.
  - `LIFOWorkList.java`: An interface representing a WorkList that stores items in LIFO order.

    - `FixedSizeFIFOWorkList.java`: An interface representing a WorkList with a fixed-size buffer that stores items in a FIFO order.
    - `PriorityWorkList.java`: An interface representing a WorkList that stores items in order of their priorities (given by compareTo).
- `egr221a.interfaces.trie`
    - `BString.java`: An interface representing a type that is made up of individual characters (examples include arrays, strings, etc.)
    - `TrieMap.java`: An interface representing an implementation of Dictionary using a trie.
    - `TrieSet.java`: An interface representing an implementation of a Set using a trie.
- `egr221a.jazzlib.*`: This is the implementation of the DEFLATE specification and Zip file io.
- `egr221a.main.*`: These are clients of the code you will be writing. Feel free t use them for testing

Implementing The `WorkLists`

In this part, you will write several implementations of the `WorkList ADT`: `ArrayStack`, `ListFIFOQueue`, `CircularArrayFIFOQueue`, and `FourHeap`. Make sure all of your `WorkLists` implement **the most specific interface possible** among the `WorkList` interfaces. These interfaces will help the user ensure correct behavior when the order of the elements in the `WorkList` matters. The `WorkList` interfaces have specific implementation requirements. Make sure to read the **Javadoc** provided HERE. (Navigate to egr221a.interfaces.worklist and read the Javadoc!)

(1) `ListFIFOQueue`

Your `ListFIFOQueue` should be **a linked list** under the hood. Again, do NOT use Java's LinkedList or List. (Not allowed to use any Java Collections). You should implement your own node class as an inner class (must be a `private static class` as in HW3) in your `ListFIFOQueue` class. All operations should be O(1). Thus you should maintain both head and tail of the internal linked list.

(2) `ArrayStack`

Your `ArrayStack` should use **an array** under the hood. The default capacity should be 10. If the array runs out of space, you should double the size of the array. When growing your array, you must do your copying "by hand" with a loop; do not use `Arrays.copyOf` or other similar methods. It is good to know that these methods exist, but for now we want to focus on understanding everything that is going on "under the covers" as we talk about efficiency. Using the length property of an array is perfectly fine. All operations should be amortized O(1).

(3) `CircularArrayFIFOQueue`

Your `CircularArrayFIFOQueue` should be **an array** under the hood. The purpose of this class is to represent a buffer that is being processed. It is essential to the later parts of the project that all of the operations be as efficient as possible. Note that there are some extra methods that a subclass of `FixedSizeFIFOWorkList` must implement.

(4) `MinFourHeap`

Your `MinFourHeap` should be an implementation of the **heap** data structure we've discussed in class. It should be **an array-based** implementation which starts at index 0. Unlike the implementation discussed in lecture, it should be a four-heap (not a two-heap). In other words, each node should have four children, not two. All operations should have the efficiencies we've discussed in class. The default capacity of the internal array should be 256. If the internal array runs out of space, you should double the size of the array.

Implementing The `Tries`

In this part, you will implement two more data structures: (5) `HashTrieMap` and (6) `HashTrieSet`.

`Tries` and `TrieMaps` Background

As briefly discussed above, a Trie is a set or dictionary which maps "strings" to some type. You should be familiar with using a HashMap and a TreeMap from EGR222 and this class. So, we'll start with a comparison to those.

Comparing `TrieMap` to `HashMap` and `TreeMap`

It helps to compare it with dictionaries you've already seen: `HashMap` and `TreeMap`. Each of these types of maps takes two generic parameters K (for the "key" type) and V (for the "value" type). It is important to understand that Tries are NOT a general purpose data structure. There is an extra restriction on the "key" type; namely, it must be made up of characters (it's tempting to think of the `Character` type here, but really, we mean any alphabet–chars, alphabetic letters, bytes, etc.). Example of types that Tries are good for include: `String, byte[], List<E>`. Examples of types that Tries cannot be used for include `int` and `Dictionary<E>`.

In our implementation of Tries, we encode this restriction in the generic type parameters:

- A: an "alphabet type". For a String, it would be `Character`. For a `byte[]`, it would be `Byte`.
- K: a "key type". We insist that all the "key types" extend `BString` which encodes exactly the restriction that there is an underlying alphabet.
- V: a "value type". There are no special restrictions on this type.

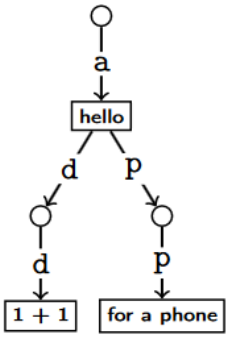
For reasons that are not worth going into, Java's implementation of generics causes us issues here. The constructor for a TrieMap takes as a parameter a Class. For our purposes, all you need to understand is that this is our way of figuring out what the type of the alphabet is. To instantiate this class, just feed in <key type class name>.class. As an example, to create a new HashTrieMap(one of the classes you will implement) which has a byte[] as the key type and a String as the value type, we would write:

```
TrieMap<Byte, ByteString, String> map = new HashTrieMap<>(ByteString.class)
```

We also use different method names from those in standard Java library: `get` becomes `find`, `put` becomes `insert`, and `remove` becomes `delete`.

(5) HashTrieMap

A HashTrieMap is an implementation of a trie where the "pointers" are made up of a HashMap. An array would work as well, but you should think about why that might not be a good idea if the alphabet size is large (like 5 or more). Consider the following TrieMap:



We could manually construct this as a HashTrieMap as follows:

```
1  this.root = new HashTrieNode();
2  this.root.pointers.put('a', new HashTrieNode("hello"));
3  this.root.pointers.get('a').pointers.put('d', new HashTrieNode());
4  this.root.pointers.get('a').pointers.get('d').pointers.put('d', new HashTrieNode("1 + 1"));
5  this.root.pointers.get('a').pointers.put('p', new HashTrieNode());
6  this.root.pointers.get('a').pointers.get('p').pointers.put('p', new HashTrieNode("for a phone"));
```

Notice that the pointers variables in each of the nodes are just standard HashMaps! You will implement all the standard Dictionary operations (insert and find). For each of these, read the **Javadoc** provided HERE for Dictionary for the particulars. (Navigate to egr221a.interfaces.misc.Dictionary) There are two methods (delete and findPrefix) which are special for TrieMaps and have additional information/restrictions:

- findPrefix(Key k) should return true iff k is a prefix of some key in the trie. For example, if "add" were a key in the trie, then: findPrefix("") = findPrefix("a") = findPrefix("ad") = findPrefix("add") = true. This method is arguably one of the major reasons to use a TrieMap over another implementation of Dictionary. (You saw a similar trade-off between HashMap (faster) and TreeMap (ordered) in previous lessons.) Unlike in a normal Dictionary, it is possible (and in fact, easy) to implement this method.
- delete(Key k) should delete k from the trie as well as all of the nodes that become unnecessary. One implementation of delete (called lazy deletion) would be to find k in the map and set its value to null (since null is not a valid value in the map). You may not implement delete as lazy deletion. Instead, you must ensure that all leaves of your trie have values. The reason we insist you write this version of deletion is that the ultimate client (zip) would be far too slow with lazy deletion.

Your implementations of insert, find, findPrefix, and delete must have time complexity Θ(d) where d is the number of letters in the key argument of these methods. These methods work on the entire key (the whole "string" of "letters"); make sure to only remove/add/find the exact key asked for.

(6) `HashTrieSet`

Now that you've implemented `HashTrieMap`, `HashTrieSet` can be implemented as a map from `K→Boolean`. In other words, sets are just a type of map! We realize that this might seem "backwards", but think about it like this: by implementing sets this way, we can avoid massive code duplication at the expense of a small amount of space. This trade-off is how it's often done in practice. The Java standard library implements `HashSet` and `TreeSet` in a similar way.

You will only need to **edit a single line of code** to implement `HashTrieSet`.

(7) Write-Up

All projects will have "write-ups" in which you answer the questions found in src/writeup/WriteUp.md. I expect fully thought out answers to the questions. These are not "throw-away points"!

(8) SuffixTrie

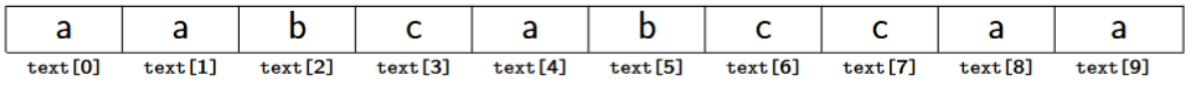
Now that you've done everything else, you should have a solid understanding of the various `WorkLists` and the idea behind a `Trie`. This last data structure, which is a type of `HashTrieSet` (but will be implemented directly as a `HashMap` with Boolean values), will use all of this knowledge. This data structure will back the LZ77Compressor which is the first phase of the compression used in zip files. It might help to skim the Wikipedia article on LZ77 (https://en.wikipedia.org/wiki/LZ77_and_LZ78) to understand the reasoning behind how this data structure works.

Tries are Prefix Trees

Another name for a trie is a "prefix tree", because they are dictionaries where looking up prefixes is easy. There are plenty of applications of standard tries: word games, predictive text, spell checking, auto-complete, etc. You will see some of these in P2 (HW8), but in P1 (HW7) we will focus on a specialization of the trie data structure called a `SuffixTrie`.

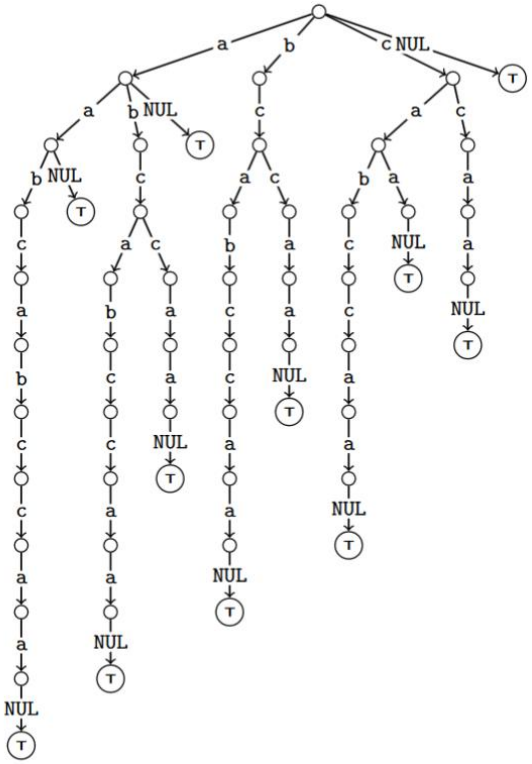
What is a `SuffixTrie`?

Imagine that we have a `String of text` which we would like to search through. The SuffixTrie for a particular String of text contains all of the suffixes of that text as entries. Consider the following example:



A `SuffixTrie` for `text` would contain the following:
{"aabcabccaa", "abcabccaa", "bcabccaa", "cabccaa", "abccaa", "bccaa", "ccaa", "caa", "aa", "a", ""}.

One (annoying) snag with a `SuffixTrie` is that we would **like every suffix to end at a leaf**. The solution to this snag is to add a "terminator" character to every suffix that actually ends the word. (Think about what the trie would look like without the `null` terminator.) We use `null` for this as follows:



Thus, to look for a word `[a1, a2, ..., an]` in a `SuffixTrie`, we look for `[a1, a2, ..., an, null]`. This data structure is useful for several reasons. First of all, it's relatively compact, because it avoids storing characters multiple times. Second of all, we can stop the search for a piece of text **as soon as the prefix doesn't match!** Another reason that this data structure is interesting is that its easy to update. Imagine that our example word **`aabcabccaa`** "shifted forward". In other words, we removed the first character ("a") and added on a new character ("x"). To update the `SuffixTrie`, all we have to do is the following:

- Remove the **key** representing the entire word. (This is just a remove on the underlying `TrieMap`.)
- For each **leaf** in the tree, remove the null terminator and replace it with a new node as picture in the right:

It might not be clear why we remove old suffixes rather than continuing to add them, and, in fact, in a different implementation, we might not remove them at all! In this implementation, we are more concerned about space than keeping track of absolutely everything, and, so, we make this design trade-off.

Implementing `SuffixTrie`

In our implementation of a `SuffixTrie`, we will represent the suffixes of a fixed-size buffer. At all times, a `SuffixTrie` must keep track of the following information:

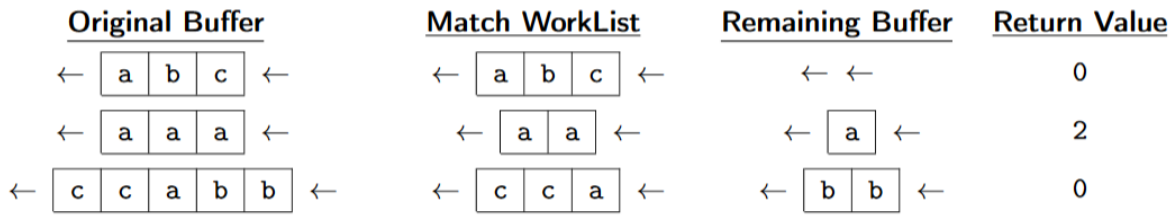
- The current match (both the letters and the node where the match ended)
- The contents that the trie represents the suffixes of (a buffer of `Bytes`)
- The current leaves of the trie (to update when the contents advance forwards

Your `SuffixTrie` must maintain this information by implementing the following interface:

`public int startNewMatch(FIFOWorkList<Byte> buffer)`

This method starts a new match from the root by consuming `Bytes` from the provided `buffer` as long as the `Bytes` consumed represent a prefix of a key in this trie. The current node pointer (i.e., `lastMatchedNode`) in the trie should also be updated. This method should return the length of the match if the match is a complete key in the trie. Otherwise, return zero. (That is, if it is a proper prefix of a key in the trie or the empty string, it should return 0.)

For example, in the trie above, the following would be the original buffer, the match consumed, and the remaining buffer:



`public void addToMatch(Byte b)`

Appends `b` to the current match. The current node pointer in the trie **should not** be updated.

`public FIFOWorkList<Byte> getMatch()`

Returns a *deep* copy of the stored match. That is, the client **should not** be able to update the field using the return value.

`public int getDistanceToLeaf()`

Returns the distance from the end of the current match to a leaf. If the match was complete, this method should return 0. Otherwise, it should return the number of (non-terminator) characters to some leaf. It is more important that this method be *efficient* than that it return a particular leaf. To do this, you should get *any* element of the pointers map. Your code to do this will look something like:

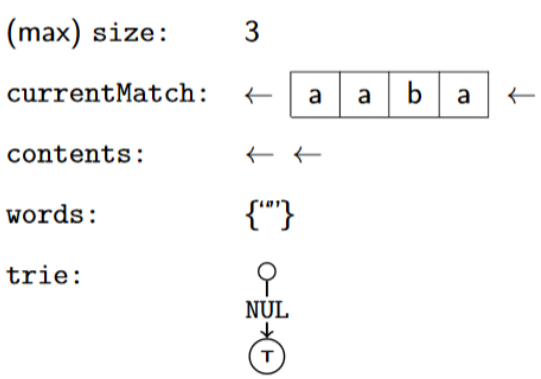
```
node.pointers.values().iterator().next()
```

`public void` **`advance`**`()`
This method advances the contents of the trie using the found match. For each `Byte b` in match, it should remove the whole word from the trie and append `b` to the end of every stored word. This is the algorithm described above to advance suffixes applied to an entire buffer. Note that if the stored contents are not yet full, we *do not shift anything off*. Note that the ordering between removal and appending (that is, removal is first) *does* matter.

`public void` **`clear`**`()`
This method should reset the state of the trie to the same as right after it was originally constructed.

An `advanced` Example

Suppose that we begin the following settings:



A single step of `advance()` (1):

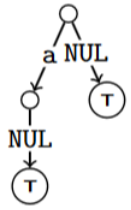
For each leaf, we replace the `NUL` with the new character (here, 'a') followed by a `NUL`. Then, we add the empty string back into the trie.

`currentMatch:` ← | a | b | a | ←

`contents:` ← | a | ←

`words:` {"a", ""}

`trie:`



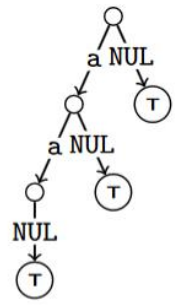
A single step of `advance()` (2):

`currentMatch:` ← | b | a | ←

`contents:` ← | a | a | ←

`words:` {"aa", "a", ""}

`trie:`



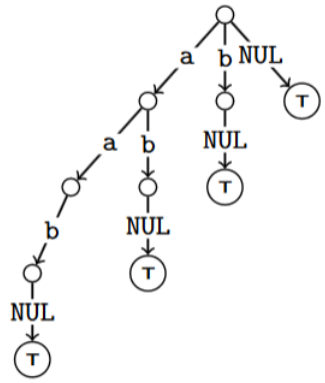
A single step of `advance()` (3):

`currentMatch:` ← | a | ←

`contents:` ← | a | a | b | ←

`words:` {"aab", "ab", "b", ""}

`trie:`



A single step of `advance()` (4):

This time we hit capacity. So, we remove "aab[null]" from the trie before extending the existing words.

`currentMatch:` ← ←

`contents:` ← | a | b | a | ←

`words:` {"aba", "ba", "a", ""}

`trie:`