CSE 332: Summer 18
Project 1: Zip

Above and Beyond

The problems in this file are not required. If you have finished Project 1 early, and would like more practice with these concepts, the following suggestions may be interesting.

Recall that any extra credit you get by completing any of these suggestions is kept in a separate gradebook. Extra credit will only be used to adjust grades near a grade boundary. It has a very small (or even no) affect on your final grade.

If you're worried about you're grade, the best use of your time by far is the main project.

`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`) 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, but in P1 we will focus on a specialization of the trie data structure called a `SuffixTrie`.

What is a `SuffixTrie`?

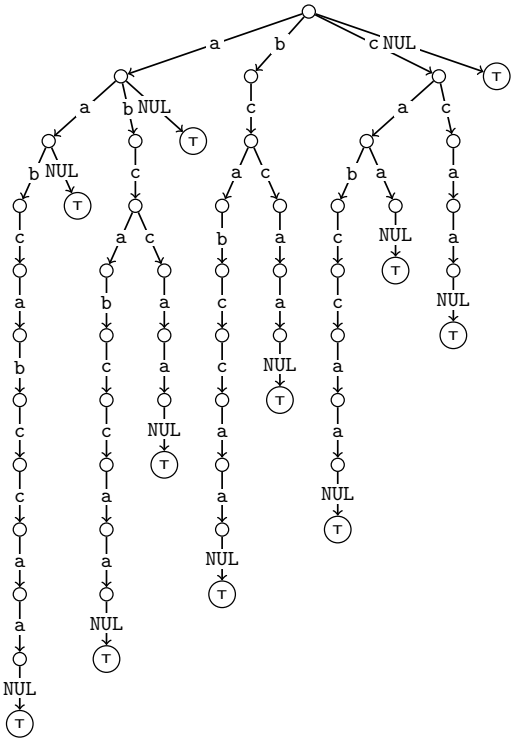
Imagine that we have a `String` of text which we would like to *search* through. The `SuffixTrie` for a particular `String` of text contains **all of the suffixes of that text** as entries. Consider the following example:

a	a	b	c	a	b	c	c	a	a
text[0]	text[1]	text[2]	text[3]	text[4]	text[5]	text[6]	text[7]	text[8]	text[9]

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 (see above and beyond for an even more compact version), 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:

It might not be clear why we remove old suffixes rather than continuing to add them, and, in fact, in a diferent 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:

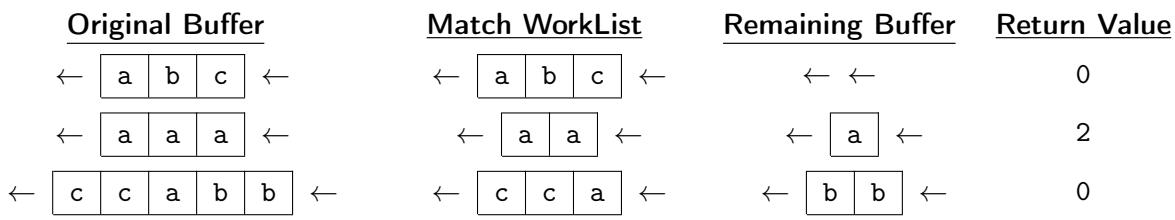
- The current match (both the letters and the node where the match ended)
- The contents that the trie represents the suffixes of (a buffer of `Bytes`)
- The current leaves of the trie (to update when the contents `advance` forwards)

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.) This method should make sure to leave room in the match buffer for one more `Byte`.

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.

See the next page for an example of `advance` in action.

An `advanced` Example

Here is an example of `advance` in action. Suppose that we begin with the following settings:

`(max) size:` 3

`currentMatch:` ← | a | a | b | a | ←

`contents:` ← ←

`words:` {“”}

`trie:`

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

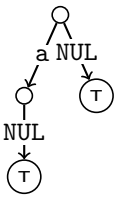
For each leaf, we replace the `NUL` with the new character (here, 'a') followed by a `NUL`. Then, we add the empty string back into the trie.

`currentMatch:` ← | a | b | a | ←

`contents:` ← | a | ←

`words:` {“a”, “”}

`trie:`



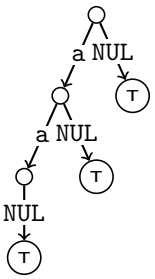
A single step of `advance()` (2):

`currentMatch:` ← | b | a | ←

`contents:` ← | a | a | ←

`words:` {“aa”, “a”, “”}

`trie:`



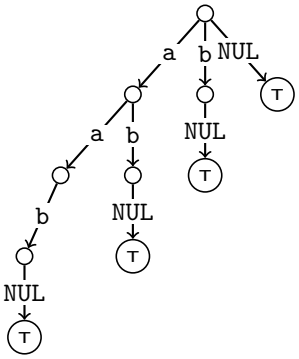
A single step of `advance()` (3):

`currentMatch:` ← | a | ←

`contents:` ← | a | a | b | ←

`words:` {“aab”, “ab”, “b”, “”}

`trie:`



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

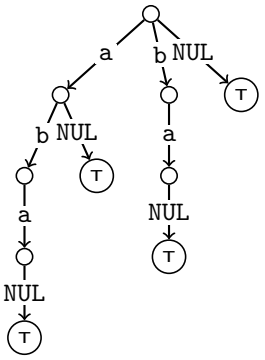
This time we hit capacity. So, we remove “aab[null]” from the trie before extending the existing words.

`currentMatch:` ← ←

`contents:` ← | a | b | a | ←

`words:` {“aba”, “ba”, “a”, “”}

`trie:`



Above and Beyond (continued)

`RandomizedWorkList`

`RandomizedWorkList` is a `WorkList` which returns all of its elements in a *random order*. This type of `WorkList` can be useful if you want a random subset of the items in the `WorkList`. For example, you could generate some random permutations to re-order the lines of a text file. We discuss two algorithms for `RandomizedWorkList`: one that doesn't work and one that does. You should think about why the Naïve Algorithm doesn't work before implementing the second algorithm.

We assume that we know *in advance* how many items the `RandomizedWorkList` will need to hold. So, it should implement the `WorkList` interface.

A Naïve Algorithm

To add the ith item **work**:

- If the buffer isn't full, add **work** to the end of the buffer.
- Otherwise, choose a random slot (each with equal probability) in the buffer and replace it with **work**

Reservoir Sampling

To add the ith item **work**:

- If the buffer isn't full, add **work** to the end of the buffer.
- Otherwise, choose a random number, j, from 0 to i. If j is a valid index in the buffer, replace the item at that index with **work**.

Client Contract

If the client calls `next()`, your implementation should throw an `IllegalStateException` on all future calls to `add`.

`CompressedHashTrieMap` and `CompressedSuffixTrie`

`CompressedHashTrieMap` is an implementation of a `HashTrieMap` which compresses together nodes that only have a single branch. For example, if the trie only had "adds" and "adam", then it's redundant to store 'd', 's', 'a', and 'm' in separate nodes. It would be better to store a single node for "ds" and a single node for "am". This will make the zip compression substantially faster if used as the underlying structure in `SuffixTrie`.