CSE 332: Data Structures and Parallelism — Winter 2021

P1: Zip

Checkpoint 1: Tuesday, January 12th
P1 Due Date: Tuesday, January 19th

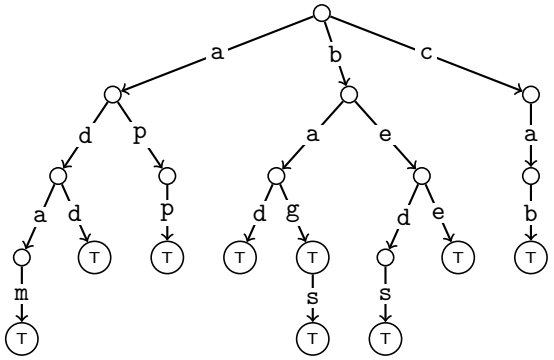
The purposes of this project are (1) to review Java, (2) to give you a taste of what CSE 332 will be like, (3) to implement various "`WorkList`" data structures, (4) to learn an important new data structure, and (5) to implement a real-world application.

Overview

A `WorkList` is a generalization of `Stacks`, `Queues`, 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). If you took CSE 143, you've actually already seen tries; you just didn't know it yet. 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. You will also be able to use these structures to compress inputs into a **.zip* file which can interoperate with the standard `zip` programs!

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, `HashTrieMap`, and `HashTrieSet` may not use any classes in `java.util.*` (including the `Arrays` class). Exceptions (like `java.util.NoSuchElementException`) **are** allowed.
- You may not edit any file in the `cse332.*` 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!

- **DO NOT MIX** any of your above and beyond files with the normal code. Before changing your code for above and beyond, copy the relevant files into the corresponding package (e.g., `aboveandbeyond`). If you choose to implement SuffixTrie you may leave that file where it currently is.
- Make sure to not duplicate fields that are in super-classes (e.g., `capacity`). This will lead to unexpected behavior and failures of tests.
- You should read through the generics handout when working on `ArrayStack`, `CircularArrayFIFOQueue`, and `HashTrieMap`. Using the methods described there will allow you to implement all of the necessary generics wrangling, but will still result in Unchecked Cast warnings in the compiler. One way to avoid these is to wrap the casting code in a private helper method and add the `@SuppressWarnings("unchecked")` annotation in the line immediately before this method.

Provided Code

- `cse332.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.
- `cse332.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`).
- `cse332.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.
- `cse332.jazzlib.*`: This is the implementation of the DEFLATE specification and Zip file io.
- `cse332.main.*`: These are clients of the code you will be writing. Feel free to use them for testing.

Project Checkpoints

This project will have a single checkpoint (and a final due date). A *checkpoint* is a check-in on a certain date to make sure you are making reasonable progress on the project. For each checkpoint, you (and your partner) will fill out a survey to track progress on the project.

As long as you submit the survey and have put forth a good-faith-effort, the checkpoint will not affect your grade in any way.

Checkpoint 1: (1), (2), (3) **Tuesday, January 12th**
P1 Due Date: (4), (5), (6) **Tuesday, January 19th**

Part 1: Implementing The `WorkLists`

In this part, you will write several implementations of the `WorkList` ADT: `ArrayStack`, `ListFIFOQueue`, and `CircularArrayFIFOQueue`. 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`.

(1) `ListFIFOQueue`

Your `ListFIFOQueue` should be a linked list under the hood. You should implement your own node class as an *inner class* in your `ListFIFOQueue` class. All operations should be $\mathcal{O}(1)$.

(2) `ArrayStack`

The default capacity of your `ArrayStack` 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 $\mathcal{O}(1)$.

(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. All operations should be amortized $\mathcal{O}(1)$.

Part 2: `Tries`

Now, you will implement another data structure: `HashTrieMap`

`Tries` and `TrieMaps`

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 CSE 143 (or equivalent). 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`.

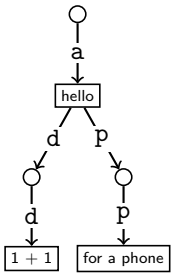
As an example, to create a new `HashTrieMap` (one of the classes you will implement) which has a `byte[]` as the key type and a `String` as the value type, we would write:

```
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`.

(4) 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` that contains a total of 6 nodes:



We could manually construct this as a `HashTrieMap` containing 6 nodes 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` for `Dictionary` for the particulars. 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 your CSE 143 equivalent.) 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 $\Theta(d)$ 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.

(5) `HashTrieSet`

Now that you've implemented `HashTrieMap`, `HashTrieSet` can be implemented as a map from `K` $\rightarrow$ `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`.

(6) `Write-Up`

All three projects will have "write-ups" which contain questions that you must answer about the project. You will find these questions here in the P1 Write-up Template. Remember to follow the instruction on the first page to receive full credit!

For p2 and p3, these will be **approximately half the project.** For p1, to get a taste of what this will be like, we will make it a much smaller portion of the total grade. Do realize though that we expect fully thought out answers to the questions. **These are not "throw-away points"!**

Above and Beyond

The following list of suggestions are meant for you to try only if you finish the requirements early. Recall that any extra credit you complete is noted and kept separate in the gradebook and may be used to adjust your grade at the end of the quarter, as detailed in the course grading policy.

`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`

`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`.