CSE 332: Data Structures and Parallelism

P1

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.

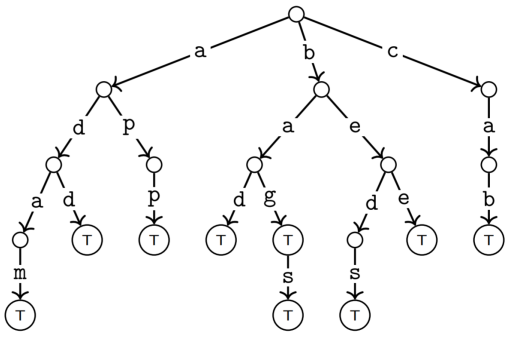
Overview

A `WorkList` is a generalization of `Stack`s, `Queue`s, 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 **`work`** to work on
`next()`	Removes and returns the next **`work`** 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).

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 `WorkList`s 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

- The *design and architecture* of your code are a *substantial* part of your grade.
- You may **not** use any classes in `java.util.*`
    - The only exception is if the code was originally there e.g. `HashMap`s in `HashTrieMap.java`.
    - 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!
- Make sure to not duplicate fields that are in super-classes (e.g., `capacity`, `root`). This will lead to unexpected behavior and failures of tests.
- You should read through the generics handout on the website 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 this 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`
  - `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 (you don't need to understand it) and Zip file io.
- `main.*`: These are clients of the code you will be writing. Feel free to use them for testing.

Project Checkpoints

Projects will have checkpoint(s) (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 will submit **code** and fill out a **survey** to track progress on the project.

Checkpoint 0: (0) due
Checkpoint 1: (1), (2), (3), (4) due
P1 Due Date: (5), (6), (7) due

5% of your total grade for Project 1 will be determined by what you submit for the checkpoints (1% for checkpoint 0 and 4% for checkpoint 1).

- For the Checkpoint 0 deadline, you will need to submit the code for Part 0 to Gradescope. (There is no survey for this checkpoint.)
- For the Checkpoint 1 deadline, you will need to 1) submit the code for Part 1 (items 1-4) to Gradescope, and also 2) fill out a survey. To receive full credit for the checkpoint you need to be passing all of the tests we have provided you for items 1-4 in Gradescope. If you are not passing all tests for items 1-4 at the time the checkpoint is due, please submit what you have at that time for partial credit for the tests you are passing. Please be sure to have items 1-4 working by the final P1 Due Date, as points will still be awarded for passing the test cases provided at that time.

Note: late days may be applied to both Checkpoint and final project deadlines.

Part 0: `HelloWorld`

Set Up

Before you begin, go through the Setting Up Your CSE 332 Environment handout. This will prepare your development environment for both this assignment and future programming projects. After you have IntelliJ and Git set up, follow the instructions on that document to clone the repository that we individually generate for everyone. You should be cloning p1-YOUR_NET_ID, instead of p1-public.

Completing the Checkpoint

You will need to change the provided method `helloWorld()` to return `"Hello World"`.

Open `src/main/java/datastructures/HelloWorld.java` to get started. You will notice that the method `helloWorld()` returns `"Welcome to CSE 332"`. This is the wrong behavior and you need to change it to return `"Hello World"`. When you are not sure about what behavior to implement, be sure to check the class's comments and any potential parent classes that it extends/implements -- this technique will become extremely important in the projects.

This repository will contain some of the testing code which will be used to autograde your project. To run the `HelloWorldTests` to check your implementation, first make sure you have set up IntelliJ and Java correctly, you can click the green Run arrow in `src/test/java/ckpt0/HelloWorldTests.java`.

Submission on Gradescope

Log into Gradescope here using your UW email. If you do not see CSE 332, let the course staff know by writing to cse332-staff@cs.washington.edu.

If this is your first time submitting a coding assignment on Gradescope, take a look at the Submitting Projects handout. Remember to use the GitLab option on Gradescope instead of uploading files directly!

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 `WorkList`s 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 $O(1)$.

(2) `ArrayStack`

The default capacity of your `ArrayStack` should be 10. If the array runs out of space, you should double the capacity 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 $\mathcal{O}(1)$.

The generics handout (that will help you create a new array of a generic type) is available here.

(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 $\mathcal{O}(1)$.

(4) Writing JUnit Tests

JUnit 5 is one of the more popular testing frameworks in Java that allows you to easily define your own unit tests and is what we use for all our tests in the projects.

Before moving onto more complex data structures, this section will cover how you can write JUnit 5 tests and using those unit tests, eventually debug your own data structures. In Office Hours, we will expect you to write your own tests to concretely understand any bugs that you're encountering, so this is an important skill to pick up!

Here is an example of a simple JUnit 5 test for `CircularArrayFIFOQueue`:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

@Test()
public void test_add_twoElements_simple() {
    int MAX_CAPACITY = 3;
    CircularArrayFIFOQueue<String> STUDENT_QUEUE = new CircularArrayFIFOQueue<>(MAX_CAPACITY);

    // Every test usually has two things:
    // (1) Get our data structure into the state that we want to be testing,
    // (2) Then, the actual test assertions.

    // First, let's set up the expected state.
    // Add two things to the queue
    STUDENT_QUEUE.add("test1");
    STUDENT_QUEUE.add("test2");
```

```
    // Then, we assert that the queue should have work.
    assertTrue(STUDENT_QUEUE.hasWork(), "Queue does not have work when it should");
    // ...but should not be full (only 2 elements added but capacity is 3).
    assertFalse(STUDENT_QUEUE.isFull(), "Queue is full when it shouldn't be");
    // The two elements at index 0 and 1 should be "test1" and "test2" respectively
    assertEquals("test1", STUDENT_QUEUE.peek(0), "test1 not found at index 0");
    assertEquals("test2", STUDENT_QUEUE.peek(1), "test2 not found at index 1");
}
```

There is a lot to unpack here but the first thing you might notice is that this is just a simple Java method with an `@Test()` annotation at the top (which is just a fancy way of telling JUnit 5 that this method is a test).

It also has a specific method name that denotes what the test is testing: test_add_twoElements_simple. That is, it is testing your `CAFQ`'s **add** method with just **2 elements** and is a simple test that tests basic methods. You should familiarize yourself with this naming convention as it will help you organize your own tests to make sure it is doing exactly what you want it to do and to also help you get a quick overview of what our tests will test.

Furthermore, there are a few JUnit 5 exclusive methods imported from `org.junit.jupiter.api.Assertions`:

Method	Description
`assertTrue(boolean condition, String message)`	Assert that the supplied `condition` is true. Fails with the supplied failure `message`.
`assertFalse(boolean condition, String message)`	Asserts that the supplied `condition` is false. Fails with the supplied failure `message`.
`assertEquals(Object expected, Object actual, String message)`	Asserts that `expected` and `actual` are equal. If both are null, they are considered equal. Fails with the supplied failure `message`.

These methods are the **basic building blocks** on writing your own JUnit tests. With just these methods and the methods you wrote for your data structures (such as `add()`), you will be able to test whether or not your data structure behaves the way you expect it to. For example, after adding "test1" and "test2" to the `CAFQ`, we assert that

`STUDENT_QUEUE.hasWork()` must return true. If it returned false, the test would have failed and printed the error message "Queue does not have work when it should".

To write your own tests, you would essentially want to take a look at the example here (or the tests we wrote in your p1 repo) and mix and match (copy and paste) these basic building blocks. The test does not have to be as complicated as the ones we have given and in fact, we encourage you to write simple tests, preferably one without complex structure like loops.

To avoid making this longer than it needs to, there will be a few other building blocks that you will encounter that are not covered here but we encourage you to reference the JUnit 5 User Guide whenever you do see them.

Some other, potentially useful methods (more in the JUnit 5 Assertions Javadocs):

Other Methods (Examples in the p1 repository)	**Description**
`assertNotNull(Object actual, String message)`	Assert that actual is not `null`.
`assertThrows(Class<T> expectedType, Executable executable, String message)`	Asserts that execution of the supplied `executable` throws an exception of the `expectedType` and returns the exception. If no exception is thrown, or if an exception of a different type is thrown, this method will fail. Fails with the supplied failure `message`.
`assertDoesNotThrow(Executable executable, String message)`	Assert that execution of the supplied `executable` does not throw any kind of exception.

Task: Finally, to get your hands dirty, we would like you to write simple tests for `CAFQ` in `YourOwnCircularArrayFIFOQueueTests.java` in the ckpt1 test directory.

Specifically, fill out the following methods in the Java file:

```
test_size_afterInsertion_incrementsByOne()
test_add_isFull_throwsException()
```

```
test_addNext_cyclesEntireQueue_returnsCorrect()
```

The Java file is also heavily documented and contains some silly test examples to ensure you understand how these tests work so be sure to read them!

✨ Congratulations on finishing the checkpoint! ✨

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 the introductory courses. 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 `Trie`s 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 -- `char`s, alphabetic letters, `bytes`, etc.). Examples of types that `Trie`s are good for include: `String`, `byte[]`, `List<E>`. Examples of types that `Trie`s **cannot be used for** include `int` and `Dictionary<E>`. In our implementation of `Trie`s, 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`.

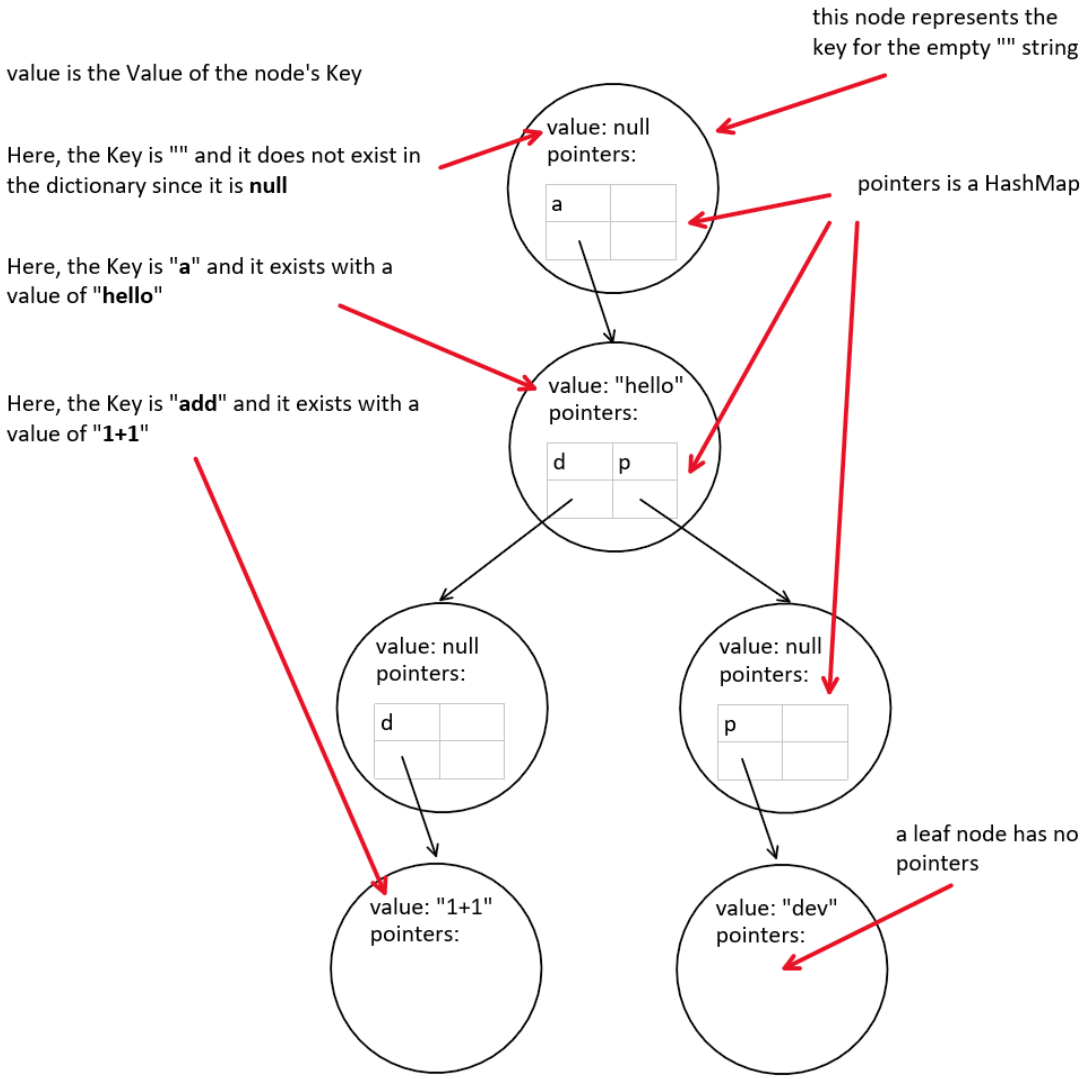
As an example, to create a new `HashTrieMap` (one of the classes you will implement) which has a `byte[]` as the key type and a `String` as the value type, we would write:

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

We also use different method names from those in the 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` that contains a total of 6 nodes:



We could manually construct this as a `HashTrieMap` containing 6 nodes as follows:

```
this.root = new HashTrieNode();
this.root.pointers.put('a', new HashTrieNode("hello"));
this.root.pointers.get('a').pointers.put('d', new HashTrieNode());
this.root.pointers.get('a').pointers.get('d').pointers.put('d', new HashTrieNode("1 + 1"));
this.root.pointers.get('a').pointers.put('p', new HashTrieNode());
this.root.pointers.get('a').pointers.get('p').pointers.put('p', new HashTrieNode("dev"));
```

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 introductory courses.) 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.

You must put your root HashTrieNode in the `this.root` field. You must also not modify the HashTrieNode class for your implementation. Our tests will be observing the internal structure in addition to the external behavior.

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.

We also discourage using your own data structures as it adds unnecessary complexity (this will be much more important later on in P2). However, if you absolutely want to, you can temporarily use built-in Java data structures until your implementation is correct and switch it out for your own data structure implementation when you are finalizing and are about to submit code (e.g. use the Java **Stack** first then your **ArrayStack**).

(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`. Before doing so, it is important to understand what the constructor of the superclass needs.

(7) Fuzzy Testing

We want to teach you that another technique for testing exists: testing using random inputs, or broadly -- fuzzing!

Normally, when you craft your own tests, you'll have to figure out what's the expected return value for some action, and write the corresponding assertions. However, remember that even though we're learning to implement our own data structures, Java also has its battle-tested implementations! For example:

- `ArrayDeque` for stack
- `LinkedList` for queue
- `ArrayList` and `LinkedList` for a normal list
- `HashMap` and `TreeMap` for map

We won't let you use these to implement your data structures, but it's fair game to use them in testing.

Here is an example of a JUnit test that always uses a "reference implementation" to determine correctness:

```
@Test()
public void testFuzz() {
    Queue<Double> reference = new ArrayDeque<>();
    FixedSizeFIFOWorkList<Double> student = new CircularArrayFIFOQueue<>(100000);
    Random randy = new Random();

    for (int i = 0; i < 100000; i++) {
        int testCase = randy.nextInt(10);
        if (testCase < 6) {
            // 60% of the time, try to insert something.
            double value = RAND.nextDouble();
            reference.add(value);
            student.add(value);
        } else if (!reference.isEmpty()) {
            // the remaining 40% of the time, try to delete something.
            assertEquals("removed element should be same",
                         reference.remove(), student.next());
        }

        // After applying the operation, we can check how things are doing.
        // use Java's built-in methods to check state.
        assertEquals(reference.size(), student.size());
        if (reference.size() > 0) {
            assertEquals(reference.peek(), student.peek());
        }

        // If you're feeling aggressive, you can also check every single value.
        List<Double> referenceList = new ArrayList<>(reference);
        for (int idx = 0; idx < reference.size(); idx++) {
            assertEquals(referenceList.get(idx), student.peek(idx));
        }
    }
}
```

Here, we alternate between inserting something and calling next() from the queue. And since everything is automated, we can test against a lot of operations (100000 here!), which is an effective way to quickly validate a hunch that there might be a bug.

This example in particular is for `CircularArrayFIFOQueue`, however, you might be able to see how you'll be able to use this for all other data structures as well! For `HashTrieMap`, you just need to generate both a key and a value, instead of just a value here, and you can validate any return values against `HashMap`.

Submission and Grading

Submission instructions can be found on the Handouts page of the website. We will grade based on correctness of code (**given** tests [some provided in code, and some that only exist on Gradescope] + **hidden** tests) and **manual grading** on whether you are following the spec. There will be **no grading** on style (such as in the introductory courses) but we suggest writing comments to help yourself understand your code and to help us grade your implementation.

Above and Beyond

The following list of suggestions are meant for you to try *for fun 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. The bottom line is that these will only have a small effect on your overall grade (possibly none if you are not on a borderline) and you should be sure you have completed the non-extra credit portions of the homework in perfect form before attempting any extra credit.

As these parts are not required, we will **not** be able to give help on them anywhere (e.g. in Office Hours or Ed).

`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 i th 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 i th 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`.