Programming Project

COP3503 Computer Science II – Summer 2023

Assignment

Implement a skip list as a genericized `SkipListSet<T>` collection in Java. Your class must implement `SortedSet<T>` (and hence `Set<T>`, `Collection<T>` and `Iterable<T>`).

A program that has your class available will be able to instantiate a `SkipListSet` for any `Comparable` class, and treat it like any other sorted collection (as long as it uses classes with a natural order).

Structure

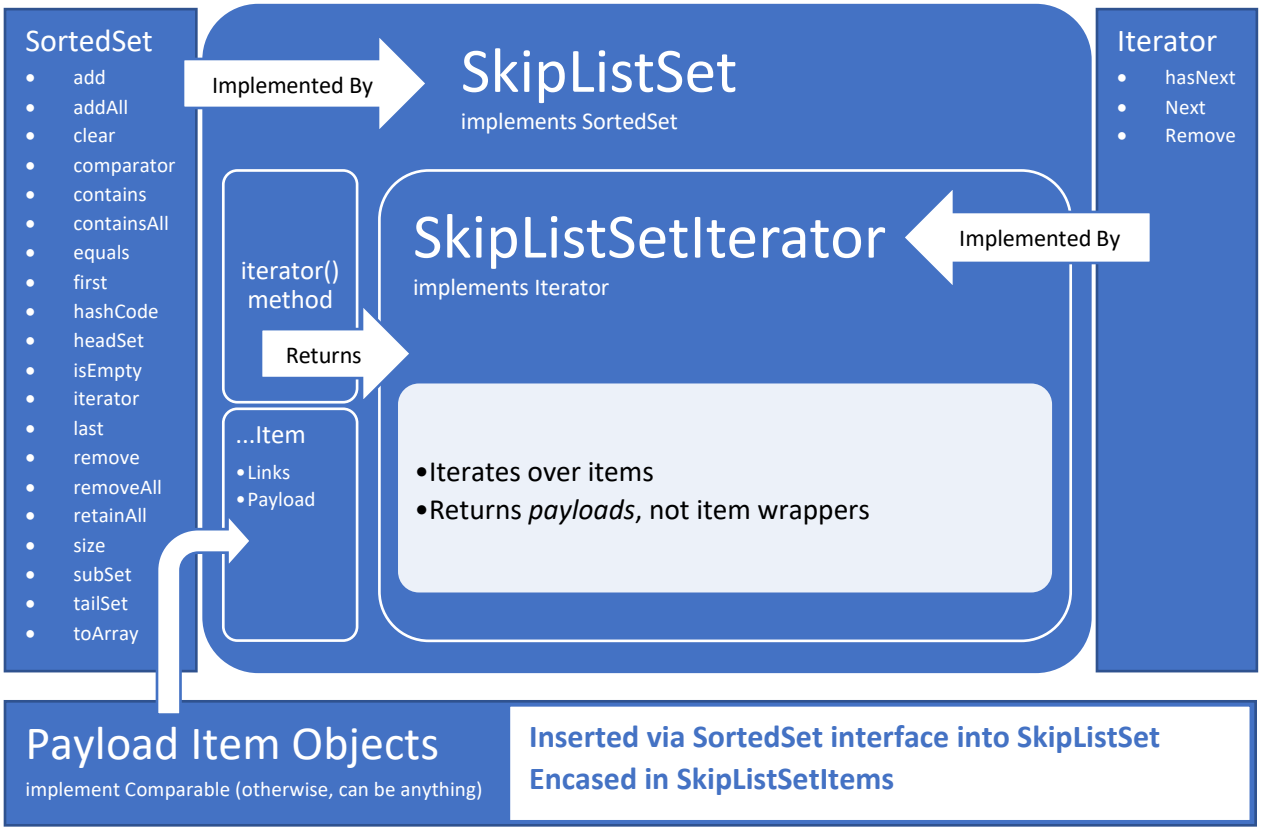
You'll create three things: a set class, an iterator class, and an item-wrapper (node) class. You'll almost certainly want to make the iterator and item-wrapper classes internal private classes of the set class. I'll call the iterator and item-wrapper classes `SkipListSetIterator` and `SkipListSetItem`, but you can call them anything you like.

`SortedSet` is a Java interface. That means it specifies a set of methods that you must implement to make it work. Your `SkipListSet` will *implement* it.

What this means is you're not calling methods from `SortedSet` yourself – you're writing them, so that a program that wants to use your `SkipListSet` can work with it the same way it would any other `SortedSet`, like `TreeSet`.

Likewise, `Iterator` is another Java interface. Your `SkipListSetIterator` will implement the methods that it requires.

Long ago in Java programming, iterators were important constructs to use directly, but enhanced-for syntax eliminates nearly all need to use them explicitly. Once you create your `SkipListSetIterator`, and implement your `SortedSet`'s `iterator()` method, you can immediately use Java enhanced-for syntax on your `SkipListSet`.



Implementing a Skip List

A skip list is a special linked list. To start with that means your item-wrapper class needs a link to the right like an ordinary linked list (and you will almost certainly want a link to the left as well).

Working with links in Java is, overall, relatively easy.

- All non-primitive parameters in Java are passed by reference as default, so you don't need to worry about pointers at all.
- Java is a mostly garbage collected language, so to delete list items, all you have to do is "lose" them – nullify all your references to them, and they'll go away.
- This means that to clear your list, all you need to do is nullify your head and tail pointers, set any internal size indicators back to zero or default, and let the garbage collector handle the rest.

A skip list is a two-dimensional linked list, so your items will need links vertically as well as horizontally. You can do this two ways:

- A true two-dimensional linked list structure, with links upward (and likely downward) from item wrappers either to other item wrappers or to an abbreviated "node hat" class that has only pointers and no payload. This is probably more efficient.
- An `ArrayList` of pointers to the right (and possibly to the left) inside each wrapper item. This is probably easier to implement.
    - (A primitive array would actually be ideal, but primitive arrays don't deal well with generics.)

No matter how you deal with the vertical dimension, you will need ways to:

- Find and traverse to the location of where an item *is* or *could/should be* in your skip list.
    - This should leverage the verticality of the skip list to be $\mathcal{O}(\log n)$.
    - All of the methods for addition, removal and containment check will probably reference this.
    - This will be distinct from your `SkipListSetIterator` class. More on this below.
- Add and delete items, and restore the left/right pointers across the whole verticality of the list.
- Shrink and grow items without breaking everything.
- Re-balance the list.

Remember that an instantiation of your item-wrapper class *wraps* an item, it is not an item *itself*. This has three important implications:

- When you're comparing items for traversal or containment checks, you want to compare the wrapped payload items, not the objects of your item wrapper class. Make sure you pass `compareTo()` through to the wrapped payload items.
- Your traversal function will be separate from your `SkipListSetIterator`. The traversal function works with the **item wrappers** that **you** need to deal with internally, while the iterator returns the **payload items** that the **user** of the skip list wants.
- If the user ever sees an item wrapper – that is, if an object of your `SkipListSetItem` class ever gets out of your list – something has gone wrong.

Working with Generics

https://docs.oracle.com/javase/tutorial/java/generics/index.html has extensive documentation about generics. *Reading this document will help with a lot of questions on generics*, and will show you several patterns you can try for individual functions.

My own implementation uses the following patterns to declare the set and its iterator, which you may free to copy:

```
public class SkipListSet <T extends Comparable<T>> implements SortedSet<T>
```

```
private class SkipListSetIterator<T extends Comparable<T>> implements Iterator<T>
```

You may suppress unchecked cast warnings for `contains()`, `remove()` and the typed version of `toArray()` – these methods all insist on accepting `Objects`, and the amount of work you'd need to do with the Reflection APIs to avoid warnings isn't worth it.

It should be possible to get the compiler happy with all the other methods. If a method is being particularly stubborn about generating warnings, feel free to ask the Teaching Assistants, or me, about it.

Algorithmic Specifics

- You can require that list items implement `Comparable`.
    - This means that you can compare your list items against each other by simply using their internal `compareTo()` methods.
- Randomize the height of the list items rather than using logarithm boundaries. 50% chance of being height 2, 25% chance of being height 3, etc.
- Implement a reBalance() method to re-randomize the height of all list items. Don't call this automatically.
- Figure out when and how to grow and shrink the maximum height of list items. (Hint: Powers of 2, and set a minimum so you don't have silly thrashing on small lists.) *Expect this to be a pain in the neck and don't leave it for the last minute.*

Implementation Checklist

You'll need to write:

- A `SkipListSet` class implementing `SortedSet`, `Set`, `Iterable` and `Collection`, described below.
- A `SkipListSetIterator` internal class implementing `Iterator`, described further below.
- A `SkipListSetItem` internal class implemented however you wish

Constructors

- Your `SkipListSet` class will need a constructor accepting no arguments, which returns an empty skip list; and a constructor accepting a `Collection` as an argument, which returns a new `SkipListSet` containing all the elements from the `Collection`.
- You don't need the other two recommended constructors from the Java collections documentation, since you may require your list items to implement `Comparable`.
- The constructors for your iterator and item classes can work however you want, since they're internal.

Methods from `SortedSet<T>`:

- `first` and `last`
- `headSet`, `subSet`, and `tailSet`, but these can simply throw `UnsupportedOperationException()` and do nothing else
- `comparator`, but it can return `null`
- You don't have to implement `spliterator`, since it has a `default` implementation

Methods from `Set<T>`:

- `add`, `addAll`, `clear`, `contains`, `containsAll`, `equals`, `hashCode`, `isEmpty`, `iterator`, `remove`, `removeAll`, `retainAll`, `size`, `toArray` (both versions)
- You don't have to implement `spliterator`, since it has a `default` implementation

Methods from `Iterable<T>`:

- You don't need to do anything here – you're already implementing `iterator` for `Set<T>`, and `forEach` and `spliterator` both have `default` implementations

Methods from `Collection<T>`:

- You don't need to do anything here – you're already implementing `iterator` for `Set<T>`, and `parallelStream`, `removeIf`, `spliterator` and `stream` all have `default` implementations

Your `SkipListSetIterator` will need to implement, from `Iterator`:

- `hasNext`, `next`, `remove`
- You *do* have to implement `remove` even though it has a `default` implementation, because its `default` implementation just throws an exception
- You do *not* have to implement `forEachRemaining`, because its `default` implementation works

Testing, Structure and Boundaries

Your `SkipListSet<T>` class shouldn't have a `main()` function. Instead, create another class for testing that does have a `main`, instantiates a `SkipListSet` and throws items at it – or work with my test harness class, and modify it however you see fit. I'll provide test cases later in the semester.

Your collection doesn't have to be thread-safe. It *does* have to work at scale – the test harness will throw a few million entries at it.

Restrictions

You may use the basic Java collections internally, but you may **not** use `ConcurrentSkipListSet` or any other existing Java skip list.

Code Conventions

In general, follow the Google Java Style Guide, with two relaxed restrictions:

- I don't care how many spaces you indent by as long as it's something reasonable.
- You don't have to use JavaDoc for methods, as long as your method comments clearly describe what the method does, what it returns, and what each parameter is for.

Submitting

Submit your code to Webcourses as a single **.java** file.