CSCI290 Project Part 02

Jemma Droppo

September 2024

EACH SUB-PART WILL NEED ITS OWN PROJECT FOLDER; THESE WILL BE MADE THE SAME WAY THAT WE DID IN THE PREVIOUS ASSIGNMENT FOR PARTS II AND III.

Part IV
Adding to our Ragged Array Lists

Objectives:

Tags

Complex Data Structures. Ragged Array List. Adding to complex data structures.

What to Expect

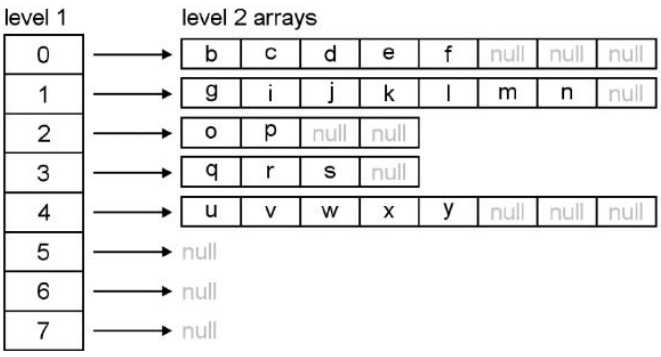

Figure 1: An example of a RAL containing letters in alphabetical order

Now we are going to write `add()` method. Your `findEnd()` methods must be working *perfectly* because you will use it to find the insertion point. For example `findEnd("h")` performed on the RAL in figure 1 would return a `ListLoc` with indices (1,1) (ab-

breviated `ListLoc(1,1)`) and then "h" would be inserted at that location. The following values "i" through "n" would all be pushed up to make room.

In order to facilitate this properly we are going to need to modify our existing conception of a ragged array list from Part III (copied into our appendixes as section 9 of this document).

More Data Structure Details:

- There will always be room to add one element into a RAL.
- If you add an item to a level 2 array, and that array becomes full, you must either split that array into 2 level arrays of the same size but each only *half* full after the split, or you double the size of that level 2 array.
- The decision of whether to split or double is based on the sizes of the level 1 and level 2 arrays. If the level 2 array is smaller than the level 1 array, then the level 2 array should be doubled, otherwise it should be split.
- For example if an "h" were added to the RAL in figure 1, its level 2 array would become full and be split into two level 2 arrays of size 8, each with four items. The level 2 arrays in indices 2 through 4 of the level 1 array would all be moved down to make room.
- If another "r" were added to the RAL in fig. 1, that level 2 array of size 4 would become full, and so it would be double to size 8 to make room for future additions.
- If you split a level 2 array, it's possible that the level 1 array becomes full, it would then be doubled.
- If things work well, as the data structure grows both the level 1 array and the level 2 arrays will be approximately the size $\sqrt{N}$ where N is the number of items stored by the RAL. (Neither the level 1 or the level 2 arrays should *ever* hold more than $\sqrt{N}$ elements; if you run into this there is something wrong with your code.)
- The initial size is 4. This is the minimum size for any RAL. Initially the level 1 array should have 4 slots and the first slot will hold a reference to a level 2 array that also contains 4 slots.

1 Setting up the Testing Code for Add

1. On Brightspace you will find a new testing file:
   - `RALtester2.jar`

2. Remove the `RALtester.jar` from the list of `.jar` files for your project.
3. Configure the build path: `File`→`Project Properties`→`Libraries`→`Run [Add JARs]`
4. The `RALtester2.jar` and `Scaffold.jar` contain code that will test your `add()` method that you will be writing for this assignment.
5. You can use the same run configuration you did for the `RALtester`. Internally `RALtester2` has the same package and class names. It just has new test cases.
6. Run the testing code. It should run and has 15 test cases.

2 Your Tasks:

2.1 Task 1: Implement `add()`

- Duplicates are allowed. When adding a duplicate item, it should always be added *after* the last matching item. That is the position that `findEnd()` finds.
- An invariant of the design is that there is always space available to insert another item. In both level 1 and level 2 arrays there is at least one unused slot. This invariant will make `add()` a little easier to write because you can trust that there is room for the insertion, and then you can deal with doubling or splitting afterwards.
- `for` loopw are not necessary for splitting. Some useful `Array` method include:
  - `Arrays.copyOf(origArray,newLength)` is useful for growing an array
  - `System.arraycopy(srcArray, srcPos, destArray, destPos, length)` is useful for inserting or splitting. `srcArray` and `destArray` *can* be the same array and it will still copy correctly.
  - `Arrays.fill(array,fromIndex,toIndex,value)` is useful for cleaning up after splitting (you can set object references to a `null` value)
- *Suggestion:* Build your solution and test incrementally! don't try to write this all at once. Do small pieces that you can test one at a time (*cough* *cough* come up with a ***unit*** and then ***test*** it). Write enough code just to insert into the L2 array and then test it. Once that is tested and working, write the code to double an L2 array and then test it. Continue this process until you've completed the process.
- Keep it simple; it is possible to do this in 25 compiled lines of code.

2.2 Task 2: Analysis of Best and Worst Cases

The ragged array list might look a bit sloppy because some rows are fuller than others (hence why they're also known as *jagged array lists*). We are going to calculate how much of a difference that could make between the best and worst cases.

1. **Best Case**: We will soon be adding 10514 songs into a ragged array list. Calculate the smallest possible size of a level 1 array that could accommodate 10514 items. Your calculation should be based on the rules for the ragged array list. Show your work and explain your answer.
   Hint: What would the array look like if elements were added such that each dimension was exactly $\sqrt{N}$ elements?

2. **Worst Case**: Now calculate that largest possible size of a level 1 array resulting from inserting 10514 items. Show your work and explain your answer.
   Hint: This worst case would occur if the items are added in a sorted order so that each new item is added onto the last level 2 array at that point during the building process.

Part V
Completing our Ragged Array Lists

Objectives:

Tags

Complex Data Structures. Ragged Array List. Custom Iterators. Adding methods: `toArray()`, `contains()`, & `subList()`.

What to Expect

Now is the time to complete the rest of the methods for our Ragged Array List: `contains()`, `toArray()`, & `subList()`, and the custom iterator for our RALs.

3 Setting up the Testing Code for a Complete RAL

1. On Brightspace you will find a new testing file:
   - `RALtester3.java`
2. Copy this into the student package of your project.
3. This is just a `.java` file so we won't need a run configuration that imports a `.jar`. We can simply select it and run it.
4. It will add 31 single character strings to your ragged array list as shown in figure 2, but your list may be different because of different choices when adding. (This will mostly depend on your choices made when writing `findEnd()`) The tester tests each of the new methods and also calculates some performance statistics.

4 Your Tasks

4.1 Task 1: Implement `contains(item)`

- Use `findFront()` (it already does the search we need).
- You can run the testing code as you implement each one of these tasks. Unlike the previous testing code this just prints the results, *you* will have to make sure they are correct.

4.2 Task 2: Implement the `Iterator` class

Note: This task may be more involved than the other tasks

- The `iterator()` method just creates an instance of the private inner class `Itr`. `Itr` implements the `Iterator` interface. All you will need to do is implement the methods `hasNext()` and `next()`. Use the provided implementation of `remove()` to satisfy the compiler.
- Internally the iterator keeps its position as a `ListLoc`. In order for the iterator to work properly you will need to implement `ListLoc`'s method `moveToNext()` which should alter the `ListLoc` to hold the index to the next item in the list.
- The method `hasNext()` only needs one line of code
- The method `next()` takes 6 lines in the answer key. Remember to throw an exception if the user goes past the end of the list.

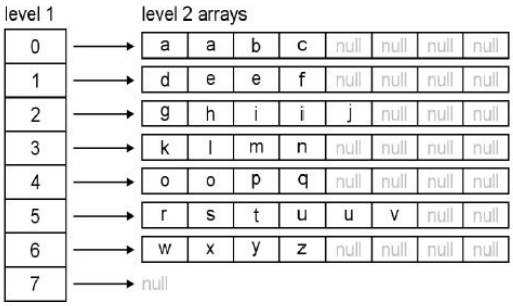

Figure 2: An example of a RAL containing letters in alphabetical order

4.3 Task 3: Implement `toArray(a)`

- This version of the `toArray()` where the caller passes in an array of the correct size. You should check for the correct size, but you don't have to reallocate a larger array or null out excess locations like the java `Collections` classes would.

- You can use the iterator from task 2 to greatly simplify writing this method.

4.4 Task 4: Implement `subList(E fromElement, E toElement)`

- The `subList()` method returns a new independent `RaggedArrayList` whose elements range from the `fromElement` (including the `fromElement`) to the `toElement` (*not* including the `toElement`). For example `subList(b,e)` on the RAL in figure 2 should return a RAL of elements `[b,c,d]`. The original list ***must*** remain unaffected, but both lists *should* refer to the ***same objects in memory***.
- This method will likely rely on `findFront()`, `ListLoc.equals()`, `ListLoc.moveToNext()`, the `RaggedArrayList()` constructor, as well as `RaggedArrayList.add()`.

4.5 Task 5: Analysis

On a `RaggedArrayList` with N items, what are the worst case times for:

1. `contains()`
2. Iterating through the whole list
3. `toArray()`
4. `subList()`

Be sure to explain and justify your answers in the associated write up.

Part VI
Search by Title Prefix using our Ragged Array List

Objectives

Tags

Complex Data Structures. Ragged Array List. Search for prefix match.

What to Expect

In this part we will be using the Ragged Array Lists we've written to *quickly* organize our songs and search them by title prefix.

YOU MUST USE THE `RaggedArrayList` ***CLASS TO DO THE SEARCHING!***

5 Your Tasks

5.1 Task 1: Create a `Song Comparator` by Title

- Add a new `Comparator` to your song class that compares songs by title.
- This comparator should ignore capitalization, similar to how the comparator for artists worked.

5.2 Task 2: `SearchByTitlePrefix`

- Create a new class `SearchByTitlePrefix`. This will be very similar to your `SearchByArtistPrefix` except that it will be using our `RaggedArrayList` class. It should have a `RaggedArrayList` field and a `Comparator` field. Most of the work is already done in `RaggedArrayList` and we will be using the data structure and its methods to support our searches.
- Your `SearchByTitlePrefix` will also need a constructor that takes in a `SongCollection` and uses the data from it to build a `RaggedArrayList` of songs ordered by title. Print statistics from the constructor after the RAL field is filled.
  Note: You are ***not*** rereading the input file! The constructor should

be using the array of songs already held in the `SongCollection` object and adding those songs to the `RaggedArrayList` field of `SearchByTitlePrefix`. This means that both the `SongCollection` and the `RaggedArrayList` field should contain references to the *same songs in memory*. This *also* means that the array of `Songs` that you get from `SongCollection` should be a *local variable*.

- Write a method called `search()` that takes in a title prefix (such as "Angel") and returns an array containing all of the matching songs. This will use the `subList()` and `toArray()` methods of the `RaggedArrayList` class.

- To do a prefix search on a range, the `fromElement` should be the first match of your title prefix and the `toElement` will be the lexicographically *next* prefix (in this case it would be "Angem"). You may need to create two dummy songs containing the 2 title prefixes for your call to `subList()`.
  Note: This can be done relatively simply and I've included a link to a GeeksforGeeks page here that comes close to the result we'd want (they ignore trailing 'z' characters).

- After the resulting RAL is built use `toArray()` to return the array of `Songs`.

5.3 Task 3: Testing

- Nearly identically to the `main()` method in `SearchByArtistPrefix`, you will need to create a `main()` method that tests your `SearchByTitlePrefix` class. It should take two arguments: the song file and the search string. It should use `SongCollection` to read the song file. It should then create a `SearchByTitlePrefix` object using the newly created `SongCollection`. Next it will need to perform the search specified by the second command line arguments. Finally it should print the total number of matches *and* the first 10 matches. Remember the code for doing this is as follows:

```
Stream.of(list).limit(10).forEach(System.out::println);
```

- In NetBeans change the `Run` configuration and add the two arguments: `allSongs.txt "search"`

- Perform searches for "search", "Angel", and "T". (The total matches for each of these should be 2, 23, and 1148 respectively.)

- Run the GUI with all 3 searches and capture the stats from the console. These will be important as the results will be included in your written report.

5.4 Task 4: Statistics

- Copy the contents of the text file `StatsMethodForPart6.txt` and place it as the last method inside your `RaggedArrayList` class (above `main()`).
- This method will allow us to verify that the code we've written for `SearchByArtistPrefix` meets our performance expectations.
- Make your new title comparator extend the `CmpCnt` class and increment `cmpCnt` each time it is called.
- In your `SearchByTitlePrefix` constructor call the `stats()` method *after* the RAL is built.
- in your `search()` method call `stats()` after the RAL is built.
- Compare these measured values to what you would expect from a mathematical analysis and explain why the measured value is or is not consistent with the mathematical analysis. Clearly explain your answer. If you chose to go for the extra credit on Part III and used a binary search for your RAL implementation be sure to explain that.

5.5 Task 5: GUI

This task is easy! Make sure the new search method automatically plugs into the GUI and works. You shouldn't need to do anything besides ensure that class properties are named correctly.

Submission

I expect your programs to be properly commented and use good style. Points may be deducted for egregious disregard of these matters. Every method in every class must have complete documentation including authorship. A full an complete revision history should be at the top of every class file! Revision comments apply only to the file they are in. Do not reference work done in other files.

All three parts will be submitted to the same assignment worth 300 points. Each part will be given up to 100 points in your final assignment grade and each parts score *will* include the associated write up.

You will submit one `.zip` archive of each project part containing any relevant code and the write up as a `.pdf`

6 Part IV

You may need to set your PDF printer as the default before running the application to print the results correctly.

6.1 Write Up (Named `Part4WriteUp.pdf`)

1. Turn in just one project report per group per part. All group members names should be listed at the top of the page.
2. Upload a pdf printout of the Report tab in the `RALtester`. Use the print button.
3. Upload a pdf of your results for `test13.txt` in the `RALtester`. Use the print button and select `Page Setup`→`Landscape` and fit to page so that it is readable.
4. Your analyses from Task 2 as outlined in section 2.2
5. It is your responsibility to test your code. Explain any incomplete parts or any known bugs. You will lose fewer points if you are aware of a bug and include it rather than leaving it unreported. ***If there are no bugs: say so!***

6.2 Grading Criteria

Your written part is complete and clearly organized	10%	`add()` is implemented completely and correctly	60%
Your analysis of the best and worst case RAL construction is thourough and complete (15% per part)	30%		

6.3 What to Submit for this Part

- `Project4WriteUp.pdf`
- `Report.pdf`
- `test13.pdf`
- `RaggedArrayList.java`

The testing PDFs can be compiled into your `Part4Writeup.pdf`, *as long as they are* ***not*** *screenshots of the test results*.

7 Part V

7.1 Write Up (Named `Part5Writeup.pdf`)

1. All group members names should be listed at the top of the page.
2. The results of `RALtester3.java` (***DO NOT SCREENSHOT***)
3. Your analyses of the worst case times for the items listed in section 4.5.
4. It is your responsibility to test your code. Explain any incomplete parts or any known bugs. You will lose fewer points if you are aware of a bug and include it rather than leaving it unreported. ***If there are no bugs: say so!***

7.2 Grading Criteria

Your written part is complete and clearly organized	10%	Task 1: `contains()` is complete and correct	10%
Task 2: `iterator()` is complete and correct	20%	Task 3: `toArray()` is complete and correct	20%
Task 4: `subList()` is complete and correct	20%	Your analysis of the worst case for each algorithm is thourough and complete (5% per part)	20%

7.3 What to Submit for this Part

- `Project5WriteUp.pdf`
- `RaggedArrayList.java` — Every method that was not written by Bob Boothe should have its author indicated in the header of the class.

8 Part VI

8.1 Write Up (Named `Part6Writeup.pdf`)

1. All group members names should be listed at the top of the page.
2. The printout of the results from the three test cases as outlined in section 5.3. (Copy and paste the console output for each of these ***DO NOT SCREENSHOT***)
3. Include your comparison count and explanation from Task 4 as outlined in section 5.4.
4. Include proof that your code works with the GUI as intended.
   You can do this by taking a screenshot of the entire GUI window with a properly searched title prefix ***not*** already searched for in the statistics section.
5. It is your responsibility to test your code. Explain any incomplete parts or any known bugs. You will lose fewer points if you are aware of a bug and include it rather than leaving it unreported. ***If there are no bugs: say so!***

8.2 Grading Criteria

Your written part is complete and clearly organized	10%	Task 1: `Song`'s comparator by title is complete and correct	10%
Task 2: `SearchByTitlePrefix` is complete and correct	30%	Task 3: proper output is given for the three test cases	20%
Task 4: Statistics and analysis are thorough and complete	20%	Task 5: Program works with GUI	10%

8.3 What to Submit for this Part

- `Part6Writeup.pdf`
- `Song.java`
- `SearchByTitlePrefix.java`

Appendix & Figures

If there are any issues with these instructions ***PLEASE*** email me! You will get extra credit for any mistakes found.

9 Appendix I: Details About (our) Ragged Array Lists

1. This is an ordered list. Items are kept in sorted order based on a comparator.
2. Duplicates *are* allowed. When searching for a match, the search should ***always*** return the earliest matching item. (E.g. if you were to add another 'b' to the above diagram, it would be inserted *between* the currently existing 'b' and 'c').
3. You will be creating a generic data structure that holds objects of a designated type (eventually they will be `Songs`, but for now we're going to be using `Strings`).
4. You will be using a `Comparator` to compare items.
5. An invariant of the design is that there is always space available to insert another item. Note in figure 1 that each of the level 2 arrays has at least one unused slot, and also that the level 1 array has at least one unused slot. This invariant will make the code *much* easier to write.

Understanding the starting code and `RaggedArralyList.java`

Spend some time looking at the stating code:

`int size`	the number of objects stored
`Object[] l1Array`	really is an L2Array[] (declared this way due to Java limitations)
`int l1NumUsed`	the number of slots used so far in `l1Array`
`Comparator<E> comp`	the comparator used for comparing items

Nested classes:

`class L2Array`	used to store the objects in the level 2 array
`class ListLoc`	used internaly to store a pair of level 1 and level 2 indices
`class Itr`	used internally to implement an iterator

Methods:

`RaggedArrayList(Comparator<E> c`	Constructor that accepts a `Comparator`
`size()`	returns the current number of items held
`clear()`	resets the data structure to empty
`findFront(E item)`	used internally to find the first match
`findEnd(E item)`	used internally to find the position *after* the last match
`add(E item)`	inserts a new item
`contains(E item)`	search for a match
`toArray(E[] a)`	copies the data structure into a basic array
`subList(E from, E to)`	returns a subset as a new ragged array list
`iterator()`	returns an iterator

The `L2Array` has:

`E[] items`	an array of items for this row
`int numUsed`	the number of items stored so far in this row
`L2Array(int capacity)`	constructor that accepts the capacity for this row

The `ListLoc` has:

`int level1Index, level2Index`	the coordinates of a location in the ragged array list
`ListLoc(int l1, int l2)`	a constructor specifying a location
`equals(Object otherListLoc` `moveToNext()`	compare two `ListLocs` move to the next item in the ragged array list.

The `Iterator` has:

`ListLoc loc`	the current position of the iterator
`Itr()`	constructor that starts at the first item
`hasNext()`	check if another item exists
`next()`	return the next item
`remove()`	not actually implemented, but needed to satisfy `implements`

Fields, internal methods, and nested classes *should* all be declared `private`, however they are declared `public` so that the code in `RALtester.jar` can actually perform the tests. Many parts of this code are already completed and marked as (done). Don't modfy those parts!

Part 1 - Parsing the Data File

Group 3: Dot Matrix and Phil A Dendron

Task 1 output from unit test of Song

```
testing getArtist: Professor B
testing getTitle: Small Steps
testing getLyrics:
Write your programs in small steps
small steps, small steps
Write your programs in small steps
Test and debug every step of the way.

testing toString:

Song 1: Professor B, "Small Steps"
Song 2: Brian Dill, "Ode to Bobby B"
Song 3: Professor B, "Debugger Love"
testing compareTo:
Song1 vs Song2 = 14
Song2 vs Song1 = -14
Song1 vs Song3 = 15
Song3 vs Song1 = -15
Song1 vs Song1 = 0
```

Task 2 output from unit test of SongCollection

```
Total songs = 10514, first songs:
Aerosmith, "Adam's Apple"
Aerosmith, "Ain't Enough"
Aerosmith, "Ain't that a Bitch"
Aerosmith, "All Your Love"
Aerosmith, "Amazing"
Aerosmith, "Angel"
Aerosmith, "Angel's Eye"
Aerosmith, "Animal Crackers"
Aerosmith, "Attitude Adjustment"
Aerosmith, "Avant Garden"
```

Task 3 output (screen capture of the GUI as described)

Incomplete Parts: None

Known Bugs: None

Figure 3: An example of what the write up for Part 01 could look like.

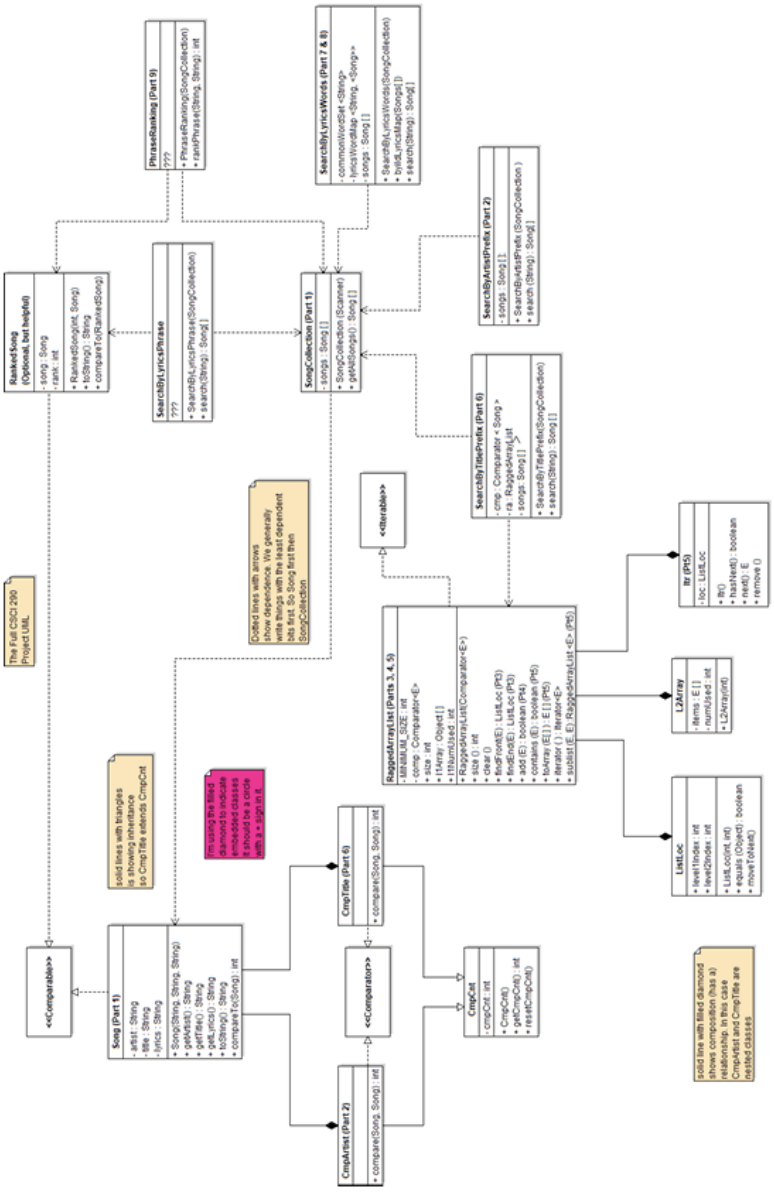

Figure 4: The full UML diagram for our completed project as it would appear in part 10