CompressionManager

Design Proposal

Sean Hinton

CSC316: Data Structures & Algorithms

sahinto2@ncsu.edu

North Carolina State University

Department of Computer Science

September 22, 2022

System Test Plan

Instructions. In this section, you must provide your system test plan with at least 5 test cases. **If you want to provide more than 5 test cases, add an appendix at the end of this document with the 6th, 7th, etc. test cases so that page numbers for all sections match the required template!**

Make sure:

- You provide your sample test data
- Test IDs are uniquely identified and descriptive
- Test descriptions are fully specified with complete inputs, specific values, and preconditions
    - Be sure to provide SPECIFIC INPUTs and VALUEs so that your test cases are repeatable
- Expected results are fully specified with specific output values
- All tests cover scenarios based on the problem statement
- All tests cover unique scenarios for the system
- All strategies for system testing are demonstrated in the tests (testing equivalence classes, testing boundary values, testing exceptions/unexpected inputs)

Test Data:

The system test plan for the CompressionManager will use the following file, `decompressed.txt`, for testing compression.

```
One fish Two fish Red fish Blue fish

Black fish Blue fish Old fish New fish
This one has a little car
This one has a little star
Say What a lot of fish there are
```

The system test plan for the CompressionManager will use the following file, `compressed.txt,` for testing decompression.

```
One fish Two 2 Red 2 Blue 2

Black 2 5 2 Old 2 New 2
This one has a little car
9 10 11 12 13 star
Say What 12 lot of 2 there are
```

The system test plan will also use the file, `empty.txt` , to test decompression and compression alternative flows when an empty file is provided

```
empty.txt has no text
```

The System test plan will also use the file `unique.txt,` to test edge cases when no compression or decompression is required while processing

```
This has only unique words

No repeats here

so no changes will be made
```

Test ID	Description	Expected Results	Actual Results

Test #1
 	
testID: testCompress
 	
Strategy: Equivalence Class - Test Loading a test file and Compressing it [UC 2]
	Preconditions:
	
	● The user has started the CompressionManager UI (UC 1)
	● The user can input the `decompressed.txt` into the UI's prompt
	
	Steps:
	
	1. **The user has started UI and selects `decompressed.txt` to compress**
	2. **The user selects the option to compress file**
	3. **UI displays the compressed version of the file `decompressed.txt`**
		The CompressionManager displays the following
		
		`Compressed Output {`
		
		` Line 1: One fish Two 2 Red 2 Blue 2`
		
		` Line 2: Black 2 5 2 Old 2 New 2`
		
		` Line 3: This one has a little car`
		
		` Line 4: 9 10 11 12 13 star`
		
		` Line 5: Say What 12 lot of 2 there are`
		`}`	

Test #2
 	
testID: testDecompress
 	
Strategy: Equivalence Class - Test Loading a text file and decompressing it [UC 3]
	Preconditions:
	
	● The user has started the CompressionManager UI (UC 1)
	● The user can input the `compressed.txt` into the UI's prompt
	
	Steps:
	
	1. **The user has started the UI and selects `compressed.txt` to decompress**
	2. **The user selects the option to decompress file**
	3. **The UI displays the decompressed version of `compressed.txt`**
		The CompressionManager displays the following
		
		`Decompressed Output {`
		
		`Line 1: One fish Two fish Red fish Blue fish`
		
		`Line 2: Black fish Blue fish Old fish New fish`
		
		`Line 3: This one has a little car`
		
		`Line 4: This one has a little star`
		
		`Line 5: Say what a lot of fish there are`
		
		`}`	

Test #3 **testID:** **testEmptyFile** **Strategy: Unexpected Input - Test handling an unexpected empty text file [UC 2,3 E 1]**	**Preconditions:** ● The user has started the CompressionManager UI ● The user can input the `empty.txt` into the UI **Steps:** 1. **The user has started the UI and selects the `empty.txt`** 2. **The user selects to compress and gets the prompt "The provided input file is empty"** 3. **The user selects to decompress and gets the prompt "The provided input file is empty"**	The CompressionManager does not display any text it instead prompts the user to use another file for compression or decompression with the message "The provided input file is empty"	

Test #4 **testID:** **testNoFile** **Strategy:** **Unexpected Input - Test an unexpected value of selecting a file that does not exist [UC 1 E 1]**	**Preconditions:** ● The user has started the CompressionManager UI ● The user can input a file that does not exist **Steps:** 1. **The user type in the location of a file that does not exist in the file explorer** a. **Eg (test-files/noFile.txt)** 2. **The UI displays a prompt to the user**	The CompressionManager reprompts the user to enter a file that does exist.	

Test #5 **testID: testQuit** **Strategy: Equivalence Class - Test that the UI properly closes the application [UC 4]** **Tests #6 and 7 in the appendix**	**Preconditions:** ● The user has selected `compressed.txt` as the input file and has started the CompressionManagerUI ● The CompressionManagerUI is currently running (UC 1) **Steps:** **1. The user has started the CompressionManager application and selected a valid file** **2. The user selects to quit the application**	CompressionManager closes and the `compressed.txt` remains the same	

Algorithm Design

Instructions. In this section, you should provide pseudocode for the specified algorithms (see the project writeup).

Make sure:

- The algorithms use the abstract data type operations as much as possible
- The algorithms contain all necessary pseudocode components to fully describe the algorithm
- The algorithms include in-line comments to explain what the algorithm is doing

Algorithm:

```
Algorithm compress(M)
    Input M, a map that represents the input line number mapped to a list of words that
appear on that line of input
    Output a Map that represents the output line number mapped to a list of words/tokens
that represent the compressed version of the input

//Initialize an empty Map to hold Strings as keys and ints as values for each entry
H <- empty Map
//Initialize an empty Map to be returned ints are the keys and Lists are values for each
//entry
R <- empty Map
order <- 1
lineNumber <- 1
for each entry l in M do
    //New List for the line of Words as Strings
    retLine <- new List
    currentLine <- l.getValue()

    //Iterate over the entire line and print the associated int if a repeat occurs
    for i <- 0 up to currentLine.size() - 1 do

        currentWord <- currentLine.get(i)
        if H.get(currentWord) is NULL then
        //If the currentWord is not associated with a key add it to the Map
            H.put(currentWord, order))
            order <- order + 1
            retLine.addLast(currentWord)


        else

            //Otherwise add the number associated with the word into the return line
            retLine.addLast(H.get(currentWord))


    //Put the compressed line into the Map to be returned
    R.put(lineNumber, retLine)
    lineNumber <- lineNumber + 1
//Return the compressed map with line numbers and their words in sorted order
sort(R)
return R
```

Data Structures

Instructions. You must determine which data structure(s) would be the best choice for implementing an efficient solution to the problem. You must:

- Describe all of the abstract data type(s) you are using
- Describe each of the data structure(s) you will use for each ADT.
- Briefly justify why you chose the data structure(s) in terms of runtime efficiency.
- If you need to sort any data, explain which sorting algorithm you will use, why you chose the specific sorting algorithm, and how you will sort your data structure(s)

ADTs:

The Abstract Data Type that will store words and its sequential number or line numbers and the Strings in the line will be a Map with the following operations

- Map consisting of entries with a Key and Value that supports getting values using keys
- put(Key, Value) Used to put unique words into the Map with their associated occurrence number or to put in a line number and its associated List of words
- get(key) Used to check if a word being read in is unique. The first occurrence of a word will return
- size(): returns the number of entries in the Map
- sort(): The map sorts itself to preserve a correct binary search
- elements(): returns an iterator of entries for iterating over every value of the input Map during compression and decompression

A Map is being used since it supports having keys with associated values. The project has line numbers with associated Lists of words that are to be compressed or decompressed then returned with the same keys and values. Another Map will also be used to compress and decompress text. In compress a Map will have unique words as keys and ints as their unique order of occurrence. In decompress a Map will have ints as keys and their unique word as value.

Maps will contain Entries containing those Keys and values and an Entry should be able to return its key (Used for searching and sorting) and return its value (Used for the Map's get method)

- getKey() Returns the Entry's key
- getValue() Returns the Entry's value

CompressionManager will also use a List to store lines of Strings for the Map that is returned from compression and decompression

- List consists of a list of words that have either been compressed or decompressed

The List will be used to store the decompressed or compressed lines in the Map returned from compress and decompress.

Required list operations

- get(index): returns the element with the associated index
- addLast(E): Adds the element to the last index in the List
- add(int, E): Adds the element E to the specified index in the List

Data Structures:

The Map will be implemented with a SkipList.

- Skip List: A Positional Linked List that supports nodes with a unique Id, up, down, left, and right pointers. Is an interconnected list of elements that searches approximately like a binary search

A skip list should be used to implement the Map ADT since every single unique word will be put into the Map and the put method requires a search of the Map for duplicate keys. Implementing the Map using a Skip List allows for time complexity of O(logn) when searching the Map during the get and put methods. Skip List should support the following methods.

- get(key): returns the value at the node with a matching key
- put(key, value): adds a new node to Skip List
- sort(): Sorts the nodes using merge sort
- search(key): Uses binary search to check if a key corresponds to an existing entry

CompressionManager will create a Map of keys and values. The keys will and values will be stored in Nodes in the Skip List and Nodes will be comparable by the value of their keys. The keys will be Strings and sorted in alphabetical order when compressing or decompressing. Numbered keys by int will be ordered in numerically ascending order. The put operation will put the new entry on the bottom level list and added to a higher level at 50% chance for every level. A Skip List should be used to implement the Map ADT since the get and put operations both use the binarySearch method so get and put will have a complexity of O(log n).

Compression and Decompression also return and take in Maps with Lists as their values.

Lists will be implemented with ArrayBasedList

Lists should be implemented using an ArrayBasedList since words will only be added to the back of the List which is O(1) amortized cost when adding to the back of the Lists for the compressed or decompressed form. The input list of words also requires get operations to check if every input word is unique or a repeat and ArrayBasedList supports O(1) accessing of the encapsulated array.

Sorting: Merge Sort

The sorting algorithm that should be used for this project is Merge Sort. Sorting will have to occur when the Map of compressed or decompressed lines is to be returned since the pseudocode should work for both ordered and unordered Maps of elements. An unordered map implementation that adds new entries to the front of the underlying data structure would put lines in reverse numerical order. Merge Sort would successfully sort this unordered Map in O(nlogn) time which is faster than other nonrecursive-comparison based algorithms that are not quick sort

Algorithm Analysis

Instructions. In this section, you should analyze the algorithm you designed earlier (see the project writeup) based on the data structures you selected.

Make sure:

- An estimated running time is provided (based on your selected data structure(s)) and matches the algorithm
- All work is shown to justify the estimated running time
- The Big-Oh growth rate is provided and is correct

Algorithm Analysis:

Algorithm:	**Analysis/Runtime Rationale**

`Algorithm compress(M)`
`    Input M, a map that represents the input line number mapped to a list of words that   appear on that line of input`
`    Output a Map that represents the output line number mapped to a list of words/tokens that represent the compressed version of the input`
 	
`//Initialize an empty Map to hold Strings as keys and ints as values for each entry`
`H <- empty Map`
`//Initialize an empty Map to be returned ints are the keys and Lists are values for each //entry`
`R <- empty Map`
`order <- 1`
`lineNumber <- 1`
`for each entry l in M do`
`    //New List for the line of Words as Strings`
`    retLine <- new List`
`    currentLine <- l.getValue()`
	n = the number of the entries in map M
	m = the number of elements in the currentLine entry from map M
	
	The outer loop repeats n times where n is the number of entries in the input Map M.

```
    //Iterate over the entire line and print the associated
int if a repeat occurs
    for i <- 0 up to currentLine.size() - 1 do

        currentWord <- currentLine.get(i)
        if H.get(currentWord) is NULL then
        //If the currentWord is not associated with a key
add it to the Map
            H.put(currentWord, order))
            order <- order + 1
            retLine.addLast(currentWord)


        else

            //Otherwise add the number associated with the
word into the return line
            retLine.addLast(H.get(currentWord))


    //Put the compressed line into the Map to be returned
    R.put(lineNumber, retLine)
    lineNumber <- lineNumber + 1
//Return the compressed map with line numbers and their
words in sorted order
sort(R)
return R
```

The inner for loop repeats m times where m is the number of elements in the current List of elements

If currentLine is an ArrayBasedList then the get operation is constant O(1) and it is completed m*n times at worst

If the underlying data structure for a Map is a SkipList then on average the get and put operation should have an average run time of O(log (m*n)) and is completed m*n times at worst

When adding to the last index of an ArrayBasedList has an amortized cost of O(1) and is completed m*n times at worst

Merge sort is used to sort the Map of compressed entries and will run in O(n log n) time because every Line is being sorted in numerical order of the lineNumber key one time before returning.

In the worst case that every single word from the Map M is unique the most costly operation get and put using map H will be completed m*n times. The compressed map must also be sorted upon completion since pseudocode does not specify Map data structure. Therefore the run time of the compress algorithm is T(n) = nlog(n) + mn log (mn) so compress is O(mn log (mn))

Software Design

Instructions. In this section, you must present your software design for implementing your proposed algorithm. You must provide a UML class diagram.

Make sure:

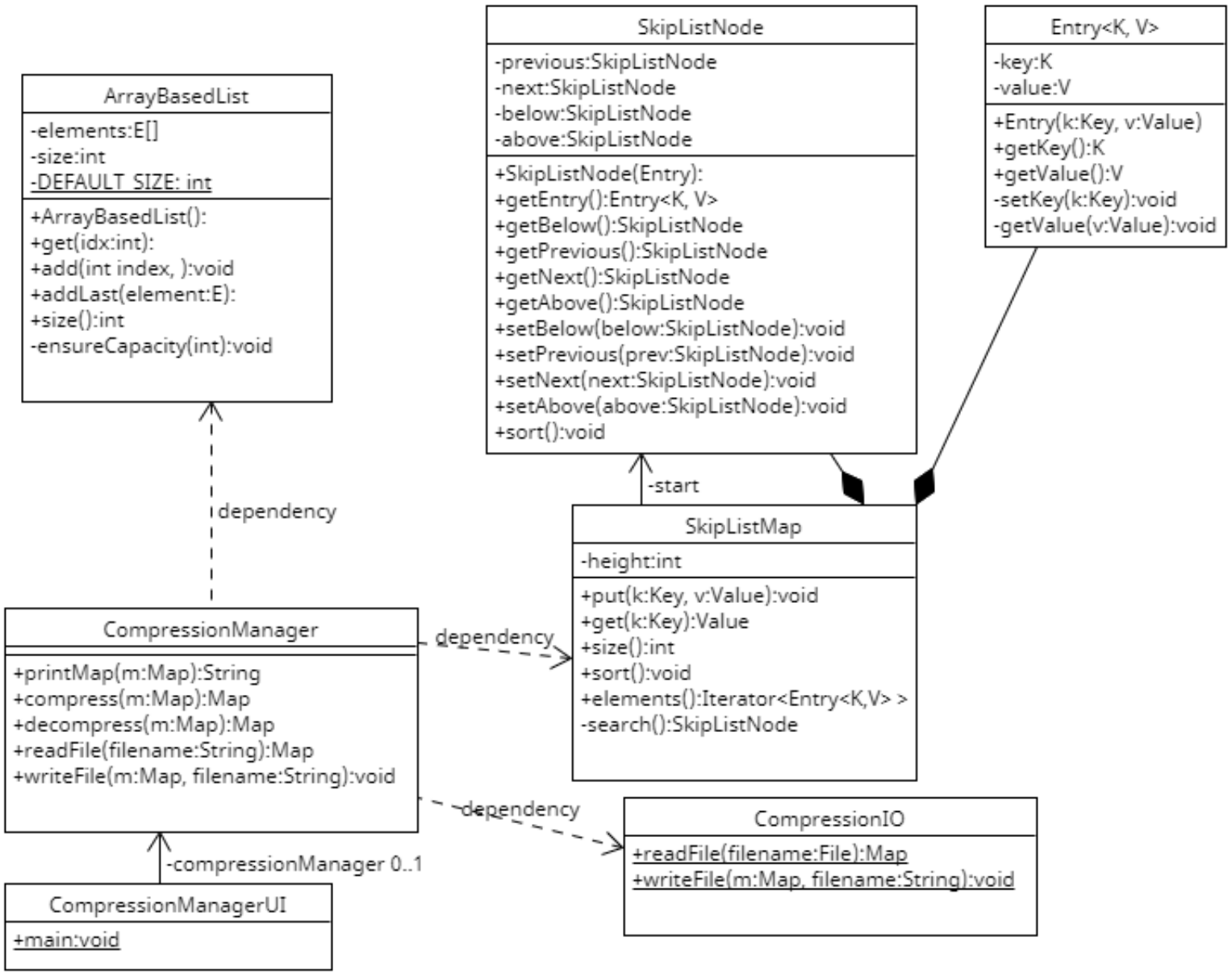
- All UML notation is correct
- All relationships required to implement the system are present
- All classes demonstrate high cohesion
- The full data structure(s) is/are included in the UML class diagram
- The UML class diagram clearly demonstrates or follows a design pattern described by the student

Brief Description:

For this software CompressionManager will use the model view controller design pattern where the CompressionManagerUI is the view and controller where the user can see what files they want to compress or decompress and have the processed outputs shown to them. The underlying model will have requests sent from the UI to the CompressionManager class which will compress or decompress the given file. There will also be a Map implemented with a SkipList that's used to hold input and output text. The SkipList will contain Entries that can get their key and value to the caller. The design will also use a custom ADT Word that represents a unique word containing the unique word's String and int value of its order of appearance. The CompressionManager will also have to work with an I/O class CompressionIO to read and write files that the user requests. Reading will take a file and produce a Map of ints and Lists of Strings while writing will take a Map and write it to a file.

Citing Workshop 2 for ArrayBasedList UML design and citing Workshop 5 for SkipListMap UML design

UML Class Diagram:



Appendix

- If any of your responses from the previous pages overflow the area we have provided in this template for you (for example, if you need more than 2 pages to present your test data from pages 1-2), then continue your responses below in this appendix. This will help ensure your PDF submission still aligns with the template that Gradescope expects.
- If you use this appendix for overflow, be sure to reference the appendix from the relevant sections within your proposals document (for example, include 'see the appendix for additional test data' at the end of page 2)
- If all of your responses from the previous pages fit within the areas provided within this template, your appendix here will be empty.

System Tests continued:

Test:	Description	Expected Results	Actual Results

Test #6
 	
testID: testNoCompression
 	
Strategy: Boundary Value - Test a boundary case where no compression is needed [UC 2]
	Preconditions:
	
	● The user has started the CompressionManager UI
	● The user can input the `constant.txt` into the UI
	
	Steps:
	
	1. **The user has started the UI and selects the** `constant.txt`
	2. **The user selects to compress the file**
	3. **The CompressionManager UI displays** `constant.txt` **with no changes**
		The CompressionManager displays the following
		
		`Compressed Output {`
		
		`Line 1: This has only unique words`
		
		`Line 2: No repeats here`
		
		`Line 3: So no changes will be made`
		`}`	

Test #7
 	
testID: testNoDecompression
 	
Strategy: Boundary Value - Test a boundary case where no compression is needed [UC 3]
	Preconditions:
	
	● The user has started the CompressionManager UI
	● The user can input the `constant.txt` into the UI
	
	Steps:
	
	1. **The user has started the UI and selects the** `constant.txt`
	2. **The user selects to decompress the file**
	3. **The CompressionManager UI displays** `constant.txt` **with no changes**
		The CompressionManager displays the following
		
		`Decompressed Output {`
		
		`Line 1: This has only unique words`
		
		`Line 2: No repeats here`
		
		`Line 3: So no changes will be made`
		`}`	