COMP 2402 W24 Lab 4 Specifications

Trees & HashSets & Maps (& Lists)

Prelab: due on brightspace by Friday March 8th, 3:00pm (no lates)
Programming: due on gradescope by Wednesday March 13th, 3:00pm (24h late ok)
Postlab: due on brightspace.ca by Wednesday March 20th, 3:00pm (no lates)
Topic focus: Lec 1 - 15

Feedback on Lab 2 Feedback

Thank you very much to the 32 of you that took the time to fill out Lab 2 feedback. I was planning on writing up a report like last time but Life got in the way. But I've carefully read over your feedback and there were some common themes:

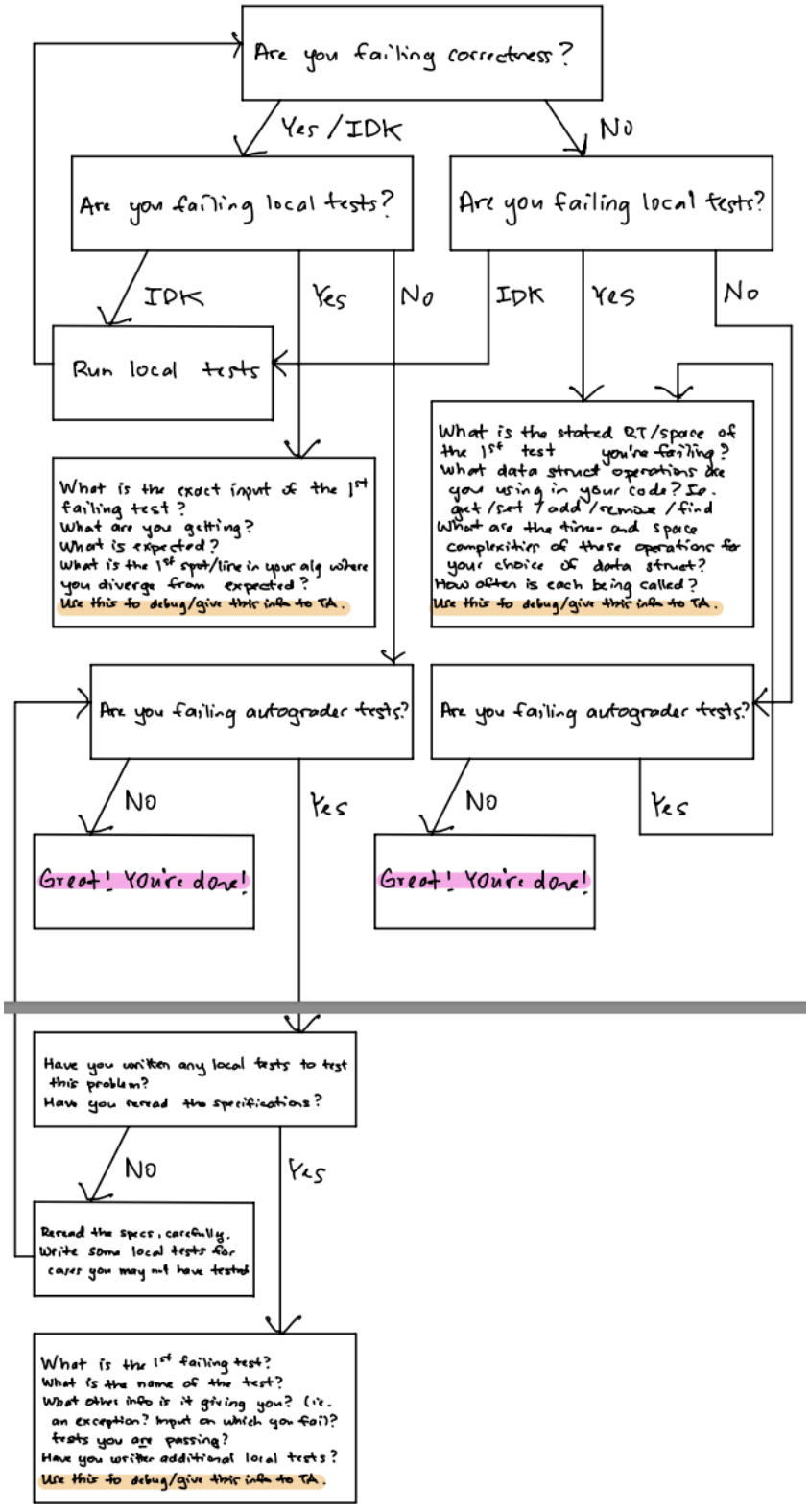
- Talk about iterators in lecture (will do next time! I mentioned their benefits in the in-person section but maybe my videos do not. Note to self to add that in.)
- For any local test written after I read the Lab 2 Feedback, you'll notice that the "Expect" comment is now in the output text as well. I hope this reduces some of your pattern matching time. Thanks for the suggestion, mystery student 🙂
- Lab 2 took longer than Lab 1 (which is natural, but maybe some warning could be given.) I can/should also emphasize how you should ask for help when you are stuck. Our TAs are on piazza all. the. time. While during "deadline week" you might not get a response right away, our average response time is 43 minutes. We've had over 600 posts in 6 weeks so we've been busy! You can't see all the private posts we get, but if you ask a question during "forum monitoring" times and you don't get a response asap it *could* be that the TA is busy responding to a bunch of other private posts.
- autograder: some issues remain although many of you appreciate getting feedback before the deadline. Without the autograder you wouldn't have any hints as to whether you had a good data structure choice and since this course is about that choice, you would have no idea whether your choice is good. But I know it can be frustrating to not know where your bugs are. Sadly, this frustration will persist throughout your computer science career in various ways. I will continue to work on naming the tests better.

Changes from Labs 1-3

The autograders have more structured feedback, so that it's a bit better at being a "checkpoint." For example, many of the PhyloTree tests are for k=1, or when the Strings are not null, etc. You can focus on these special cases first and then worry about more general cases, if you wish.

There are still no hidden tests. In a sense, the autograder variability serves as a "hidden test" in that you sometimes have to look at your code and determine its complexity (gasp!) and decide whether it matches the desired complexity rather than relying on the occasionally flaky tests.

Finally, I made this new flow chart to help you determine where your problems are and how to communicate those problems to the course staff so that we can help you as quickly as possible!



Lab Objectives

The labs in this course are meant to give you the opportunity to practice with the topics of this course in a way that is challenging yet also manageable. At times you may struggle and at others it may seem more straight-forward; keep trying and practicing and you will improve.

Specifically, Lab 4 aims to improve your

1. Implementation Skills
   Implement iterative operations for tree-based data structures similar to the BinaryTree and BinarySearchTree (add, find, remove, iterate, traverse, rotations).
2. Design Skills
   Use object-oriented programming concepts such as abstraction and polymorphism to provide flexibility in your choices of which interface implementation to use.
3. Critical Thinking
   Demonstrate a solid understanding of the pros and cons of various implementations of the SSet (randomized, expected, worst-case) and of the situations when an unsorted set or map might be a better choice than a sorted set or list. This includes considering the time- and space-complexity of various operations in different data structures.
4. Algorithmic Thinking
   Apply algorithmic thinking to design efficient algorithms for common tree-based operations, such as find(x), iteration over nodes using pointers, careful pointer manipulation in tree rotations, and removing recursion to avoid a big call stack.
5. Error Handling
   Implement appropriate error handling and boundary checks to ensure the robustness of the data structures.
6. Testing
   Determine the necessary tests to ensure your algorithms are correct and efficient.
7. Planning
   Maintain or adopt good academic and programming habits through the pre-lab.
8. Reflection
   Reflect on your choice of data structures and algorithms through the post-lab.

Assignment Components

Details of each component follow later in the specifications.

1. (6.7 points) Prelab (complete on brightspace)
2. (80 points) Programming portion (submit on gradescope.ca)
   a. (50 points) `PhyloTree` Implementation
   b. (10 points) `Genes` implementation
   c. (10 points) `GenesOrder` Implementation
   d. (10 points) `GenesCount` implementation
3. (13.3 points) Postlab (complete on brightspace)

Grading Criteria & Submission Guidelines

See the Grading Criteria and Submission Guidelines of Lab 1; they are the same.

Collaboration & Academic Integrity

1. Individual work is expected. Any collaboration should be explicitly mentioned and acknowledged at the top of each file. It is **okay to discuss high-level approaches** with **your peers, or low-level syntax-type questions**, but you must construct your solution on your own (as in: you have to formulate the *code* of your solution on your own.) Consider the analogy of writing an essay. You might talk with a peer about the high-level concepts of your thesis, or you might ask them about grammar or even phrasing of individual sentences. But you should not be writing the essay sentence-by-sentence with someone else's help; in the end you have to sit down and write that thing on your own.
2. Plagiarism will result in severe consequences. **Ensure that all code and documentation are your own work**. Do not send code to or receive code from any source except for course staff or the textbook, even if you change a thing here or there. It helps to keep the analogy of an essay in mind; it is not okay to take a paragraph from a friend and then rearrange some of the words or replace some with a thesaurus. It's not even okay to paraphrase each sentence. It is not okay to send your essay to a peer. Similarly here, you cannot start with code that is not yours and then "make it your own" with minor edits. Automated tools for detecting plagiarism will be employed in this course.
3. The same restrictions apply to **AI programmers** (such as chatGPT, copilot). You can use them to help with basic syntax (e.g. "spelling" and "grammar") or to understand broad concepts (e.g. getting feedback on a thesis) but you have to formulate your solution in code on your own (e.g. you have to write that essay yourself.)
4. Note that **contract cheating sites** are known, unauthorized, and regularly monitored. Some of these services employ misleading advertising practices and have a high risk of blackmail and extortion.
5. Every student should be familiar with the Carleton University student academic integrity policy. Academic integrity is upheld in this course to the best of Prof Alexa's abilities, as it protects the students that put in the effort to work on coursework within the allowable parameters. Potential violations must be reported to the Dean of Academic Integrity. If you ever have **questions** about what is or is not allowable regarding academic integrity, **please do not hesitate to reach out to course staff**. We are happy to answer.

Copyright

Workflow

In a perfect world, this is how you would complete Lab 4:

1. Attend or watch the relevant lectures that are listed in the heading of this document.
2. Read the lab objectives listed on the second page of this document. For each data structure listed there, review its important algorithms as well as the time- and space-complexity of its methods. This should give you some pros and cons for each.
3. Carefully read each problem detailed in the Programming section of this document.
    a. Make sure you understand the problem.
    b. Try the given examples by hand to get a better understanding of the problem.
    c. Pay special attention to any special cases or edge cases.
    d. Try more examples of your own devising if you need them.
    e. Do not start programming yet!
    f. Consider attending or watching the lab's workshop video, posted on brightspace.
4. Once you have completed steps 1-3, you are ready to do the prelab (brightspace).
5. Complete the programming portion of the lab.
    a. Take this one problem at a time. Any order should be okay.
    b. Remember what you learned in Steps 1-4 as you brainstorm solutions.
    c. Test locally at frequent intervals. Do not write a whole program then test it afterwards. See the section on Local Tests to help you here.
    d. Submit to gradescope.ca whenever you have made good progress, but do not use gradescope.ca as your only tests. Gradescope keeps your most recent score unless you select a different submission to be active. See the section on the Gradescope Autograder to help you debug here.
    e. If you're stuck on a problem for more than 30 minutes, ask for help using the How to Get Help section. Move on to something else until help has arrived.
6. Once the late programming deadline has passed, complete the post-lab (brightspace).
    a. If you did well on the programming portion, this is not meant to take too long.
    b. There are resources available to help you posted on brightspace under the Lab 2 module. There is a solutions walk-through video for each part, and also a debrief document where I walk through the problem solving process and learning engagement I was hoping you would experience. You might consider looking at these before completing the prelab if you had trouble with any of the programming parts.
    c. You do not have to have completed the programming parts in order to do the postlab. If you were stuck on a problem, this is an opportunity to look at the sample solution videos, to figure out what went wrong for you, and to still learn what you were meant to learn.

Coding Environment Setup

Lab 1's Coding Environment Setup will work with Lab 4 (replace `Lab1/l1` with `Lab4/l4`). The file structure and many of the files will be the same, with new and different files as well.

Programming Components

Programming Notes

1. You may use java.util.HashSet instead of ods.MultiplicativeHashSet if you want the fastest implementation of a HashSet. Note that HashSet has an iterator.
2. You may use java.util.HashMap as well, for a fast implementation of a HashMap. Note that HashMap has an iterator over its keys or values.
3. If you want to use Collections.sort on an array-based Collection, you can!

   e.g.
   ```
   ArrayDeque<String> ad = …     // initialize ad etc
   Collections.sort(ad);         // now ad is sorted!
   ```
   This takes O(n log n) time, where n is the number of elements in `ad`.
   You can also sort a Collection using a new comparison rule (a Comparator) , eg

   e.g. `Collections.sort(ad,MyComp); // sort ad using MyComp`

   where you may have defined `MyComp` elsewhere like this
   ```
   class MyComp implements Comparator<String> {
        public int compare(String a, String b) {
             // Fill in compare
        }
   }
   ```
   Here are some examples on geeksforgeeks. Note that sorting a linked-list-based Collection can be done, but java first dumps the linked list into a structure with fast random access, and then sorts on that better structure, then dumps the sorted thing back into your linked-list-based structure. Keep that in mind, I suppose.
4. If you want to store objects of a class (of your own definition) in a sorted set, you need to ensure your class implements Comparable. For example, you could have
   ```
   class MyClass implements Comparable<MyClass> {
        int x;
        int y;
        public int compareTo(MyClass b) {
             // Fill in compareTo
        }
   }
   ```
   Here are some examples on geeksforgeeks.
5. Note that you should not need recursion for any of the problems. You can/should run all tests with the `-Xss144k` flag (or, the smallest heap size your local machine will allow.) Sometimes you'll pass such tests even with recursion.

Implementation Practice [50 marks]

PhyloTree [50 marks]

`PhyloTree` represents a tree of distinct Strings (where a String can ostensibly represent a DNA sequence; no biology knowledge necessary for the lab). Your task is to implement the described methods according to the specifications below, using the starter code and lecture to guide you. The provided implementations are incomplete and incorrect, but should be enough to compile and run the (failing) local and autograder tests.

The data structure is a modified BinaryTree of Strings of some fixed length `k`, where each String is either `null` or of some length `k>0` that is fixed for the given tree. The tree represents a "phylogenetic tree," which is essentially a hypothesized family tree for DNA. When a node has value `null` it means we haven't determined that ancestor in the tree yet. Each (non-null) String of the tree should be distinct and of length `k`.

This is the general idea. Let's get to some details.

Inner Class

`Node` (Represents a Node in our tree.)

`String`	`s`	The `String` of length `k` in this `Node`, or `null`.
`Node`	`parent, left, right`	The parent, left child and right child of this Node.
`boolean`	`mark`	Helpful node-specific boolean, used in `LCA`.
`boolean[]`	`set`	Helpful node-specific set, used in `computeSet`.

Fields

- `Node r` — The root `Node` for the tree.
- `int n` — The number of nodes in the tree.
- `int k` — The length of non-null Strings in the tree, `k>0`.
- `HashMap<String,Node> stringsToNodes` — A map from each non-null String stored in the Tree to the node that contains that String. Used to speed up various methods.

Implemented Methods & Constructor

- `PhyloTree()`
  Initializes the fields so that this tree is empty and `k = 1`.

- `PhyloTree(int k)`
  Initializes the fields so that this tree is empty and initializes `k` to the parameter.

- `public int size()`
  Returns the number of nodes in the tree.

- `public void clear()`
  Clears the tree so that it contains no elements.

- `public String toString()`
  Returns a list-based `String` representation of the tree in in-order that is helpful for debugging purposes.

- `public String prettyPrint()`
  Returns a tree-like `String` representation of the tree that is helpful for debugging purposes. This is only useful for smaller trees; for larger trees with missing children it can misrepresent the tree so use it cautiously.

- `public int height()`
  Returns the first `Node` in an in-order traversal. Note that his is a recursive method so use it with caution.

- `protected Node firstNode()`
  Returns the first `Node` in an in-order traversal.

- `protected Node nextNode(Node w)`
  Returns the `Node` that follows `w` in an in-order traversal.

- `public Iterator<String> iterator()`
  Returns an `Iterator` that iterates over the Strings of the tree in in-order order. Note that the Strings are not being stored in sorted order, so it may not appear sorted.

Constructor & Methods For You to Implement

Implement all the following methods without recursion.
Don't forget to update `stringsToNodes` when you add a non-null String to the PhyloTree!
Use the examples from lecture (e.g. `traverse2`, `size2`, `findLastNode`) to guide you here.

- `public boolean addChild(String parent, String child) throws IllegalArgumentException`
  Adds a `Node` containing `child` as the child of the existing `parent`.
  If `parent` is null, adds `child` as new root, with existing root as its left child.

If `parent` has no children, add `child` as a left child.
If `parent` has left child, add `child` as a right child.
Throws an `IllegalArgumentException` if `parent` already has 2 children.
Throws an `IllegalArgumentException` if `parent` is not `null` and doesn't exist.
Throws an `IllegalArgumentException` if `child` is not `null` and already exists
Throws an `IllegalArgumentException` if `child` is not `null` and its length is not `k`.

If you've implemented `addChild`, it is a very good idea to test this locally and on the autograder before proceeding. If your `addChild` isn't working, none of the other tests are likely to work.

- `public PhyloTree(ArrayStack<String> a)  // Constructor`
  Initializes the tree to contain the elements of `a`, where `a` contains the elements of the tree in "level order" (i.e. `a[0]` is the root, `a[1]` and `a[2]` are the root's children, and so forth.)
  Throws an `IllegalArgumentException` if non-`null` duplicates exist.
  Throws an `IllegalArgumentException` if any non-`null` String has length != `k`, where `k` is the length of the (non-null) Strings in `a` (they should all have same length.)

If you've implemented `PhyloTree`, it is a very good idea to test this locally and on the autograder before proceeding. If this constructor isn't working, many of the tests that depend on this constructor will also fail.

- `public String LCA(String s, String t)`
  Returns the "least common ancestor" (LCA) of the nodes containing `s` and `t`.
  The LCA is the first ancestor that `s` and `t` have in common (the one farthest from root `r`).
  Throws an `IllegalArgumentException` if `s` or `t` are `null` or aren't in the tree.

  Note: The `Node` class has a `mark` field that can help you keep track of, say, nodes you've seen. Just remember that if you "mark" a node, you should "unmark" it before you return from LCA otherwise it will be "marked" the next time you call it.

  Hint: break your code into manageable parts. Perhaps separate out the logic of checking for exceptions from finding the nodes that contain `s` and `t` from the logic that find the LCA of those nodes.

- `public void fixUp(String s)`
  Performs a series of single rotations involving `s` until `s` is in a "good spot."
  In particular, if the node containing `s` is not in "binary search tree" order relative to its parent (e.g. it's a left child greater than its parent, or a right child less than its parent), use the appropriate single rotation to swap the parent-child relationship.
  Proceed up the tree as necessary until `s` is "good" relative to its parent.

If `s`'s parent is `null`, it is in a "good spot".
Note that the rest of the tree may be completely disordered; that's okay.
Throws an `IllegalArgumentException` if `s` is `null` or is not in the tree.

Hint: single rotations come up in Treaps and BinaryHeaps; look there for guidance.

- `public void computeSets(int index)`
  For this problem, each `Node` can hold a String of length `k` of `DNABases` (or is `null`).
  (You can use the provided class field `DNABases = {A,C,G,T}` here.)
  Initially, all the leaves will be non-`null`; the internal nodes may or may not be `null`.
  The given `index` indicates which index of the length-`k` String is relevant.
  This problem asks you to compute each node's set of possible bases for the given index.
  You'll use the node's `set` field for this:
    i.e. if `DNABases[b]` is a possible base for the given index of node u, you'll set
      `u.set[b] = true`
    (ex: if (index `index`) of `u` can be `A,G` then `u.set=[true,false,true,false]`)
  Throws an `IllegalArgumentException` if the `index<0` or `index>=k`.
  Throws an `IllegalArgumentException` if any of the leaves are `null`.

  Use the following algorithm to compute the sets:

```
for each node v in post-order (working from leaves upwards)
     if v is a leaf
          v's base is v's character at given index
     else if v has one child c
          v's set of bases is c's set of bases
     else
          if v's left and right children have overlapping sets
               v's set= the overlap/intersection of left &
          right
          else (the sets do not overlap)
               v's set = the combination/union of left & right
```

  Note: there are some methods for debugging and testing that will be helpful:
  `setToBases(Node u)` - returns a String representation of node u's set
  `allSets()` - returns an ArrayStack of all node's u's sets, in post-order

Desired Complexity

In the following table, let

- `n` denote the number of nodes in our tree;
- `d(s,t)` denote the length of the path between nodes `s` and `t`
- `rotations` denotes the maximum number of rotations needed to fix u (which in the worst-case could be the length of the `root → u` path, but in best case could be O(1))

marks	method	time complexity	(extra) space complexity
10	add(p,c)	O(1)[E]	O(1)
10	PhyloTree(a)	O(n)	O(n)
10	LCA(s,t)	O(d(s,t))	O(1)
10	fixUp(u)	O(rotations)	O(1)
10	computeSets(index)	O(n)	O(1)

Examples

addChild

Tree (pre)	k	method	Tree(post)
A	1	addChild(A,B)	A B
A B	1	addChild(A,C)	A B C
A B	1	addChild(B,C)	A B C
AA BB CC	2	addChild(CC,DD)	AA BB CC DD
AA BB CC	2	addChild(CC,null)	AA BB CC null
A B C	1	addChild(null,D)	D A B C
A B C	1	addChild(A,D)	IllegalArgumentException (parent already has 2 children)
A B C	1	addChild(B,C)	IllegalArgumentException (child already exists)
A B C	1	addChild(D,E)	IllegalArgumentException (parent not found)

A B C	1	addChild(B,DD)	IllegalArgumentException (child is not of length k)

PhyloTree

method	Tree (post)	k	n
PhyloTree(A,B)	A B	1	2
PhyloTree(B,A)	B A	1	2
PhyloTree(A,B,C,D)	A B C D	1	4
PhyloTree(null,null,AA, BB, CC)	null null AA BB CC	2	5
PhyloTree(A,B,C,D,E,F,G,H,null)	A B C D E F G H null	1	9
PhyloTree(A,A)	IllegalArgumentException (duplicate element}		
PhyloTree(A,BB)	IllegalArgumentException (A and BB have different lengths)		

LCA

Tree	k	method	output
A	1	LCA(A,A)	A
A B	1	LCA(A,B)	A
A B	1	LCA(B,B)	B
AA BB CC	2	LCA(BB,CC)	AA
A B C	1	LCA(D,C)	A

D E			
A B C D E	1	LCA(D,E)	B
A B C D E	1	LCA(E,A)	A
null B C D E	1	LCA(E,C)	null
A B C	1	LCA(A,D)	IllegalArgumentException (input D not in the tree)
A B C	1	LCA(D,null)	IllegalArgumentException (input cannot be null)
A B C	1	LCA(B,DD)	IllegalArgumentException (input DD is not in the tree)

fixUp

Tree (pre)	**method**	**Tree(post)**
A B	fixUp(A)	A B
null B	fixUp(B)	null B
A B	fixUp(B)	B A
A B C	fixUp(B)	B C A
A B C	fixUp(C)	C A B
C B A	fixUp(A)	A C B
D A B C	fixUp(C)	C D B A

A B C	fixUp(null)	IllegalArgumentException (input is null)
A B C	fixUp(D)	IllegalArgumentException (input not in tree)
A B C	addChild(D,E)	IllegalArgumentException (parent not found)
A B C	addChild(B,DD)	IllegalArgumentException (child is not of length k)

computeSets

Tree (pre)	**k**	**method**	**Tree with Sets**
AT	2	computeSets(0)	AT[A]
AT	2	computeSets(1)	AT[T]
null AC AG	2	computeSets(0)	null[A] (intersection) AC[A] AG[A]
null A C	1	computeSets(0)	null[A,C] (union) A[A] C[C]
TT CC GG AC CA AT GT	2	computeSets(0)	TT[A] CC[A,C] GG[A,G] AC[A] CA[C] AT[A] GT[G]
A C	1	computeSets(0)	A[C] C[C]
A G T	1	computeSets(1)	IllegalArgumentException (index >= k)
A G null	1	computeSets(0)	IllegalArgumentException (leaf is null)

Interface Practice [30 marks]

Genes, Revisited

In lab 3 the Genes problem asked you to compute the number of different strands of length k encountered in the input. This problem is similar in many ways, so I recommend you first review a sample solution (yours or ours) or the debrief before tackling this problem.

Method Signature

```
public static int genes(InputGenerator<Character> gen, int k)
```

Method Behaviour

Decodes each gene (substrand) of length $k > 0$ in the sequence generated by a given `InputGenerator` as follows, then returns the number of different decodings:

For gene consisting of input characters $c_i c_{i+1} c_{i+2} \ldots c_{i+k-1}$ we decode it as

$$i \cdot c_i + (i+1) \cdot c_{i+1} + (i+2) \cdot c_{i+2} + \ldots + (i+k-1) \cdot c_{i+k-1}$$

Desired Complexity & Notes

$O(n)^E$ time, where n is the number of characters generated.

$O(k+d)^E$ space, where k is the integer input parameter, and d is the solution.

- Since there are 4 bases, $d \leq 4^k$ and thus $\log d = O(k)$.
- For reference, in Java, A=65, C=67, G=71, U=85.
- For large n and k, it's conceivable your decoding will have integer overflow. That's okay, let it overflow. The tests will overflow as well, which is simpler than describing ways to prevent overflow.

Examples

characters generated by `gen`	`k`	decoding of individual characters	different decodings	output (`d`)	`n`
AAAA	3	[A·0,A·1,A·2,A·3] [0,65,130,195]	[195,390]	2	4
AAAAU	3	[0,65,130,195,340]	[195,390,665]	3	5
UGCGAAUAUA	3	[0,71,134,213,260,325,510,455,680,585]	[205,418,607,798,1095,1290,1645,1720]	8	10
UUUUCCCCAAAA	12	[0,85,170,255,268,335,402,469,520,585,650,715]	[4454]	1	12

`UUUUCCCCAAA`	12	`[0,85,170,255,268,335,402,469,520,585,650]`	`[]`	`0`	11
`... c`$_{64}$`CGAc`$_{68}$`...`	1	`[..,` `C·65,G·66,A·67,..]`	`[..., 4355, ...]`	`<n`	`n`
(no chars)	> 0		`[]`	`0`	0

Testing & Autograder

There are limited local tests in `Genes.main` and `tests/GenesTest.java`; see Lab 1 for testing instructions. Submit `Genes.java` to gradescope; see Lab 1 for submission instructions.

GenesOrder, Revisited

Method Signature

```
public static AbstractList<String> genesOrder(InputGenerator<Character>
gen, int k)
```

Method Behaviour & Notes

Returns a list of the different genes (substrands) of length `k > 0` in the sequence generated by a given `InputGenerator`, in sorted order.

An AbstractList is an abstract java interface that is implemented by all the ods List implementations (e.g. ArrayStack, ArrayDeque, DLList, SkiplistList, etc). Using AbstractList<String> as the return type means you have the flexibility to choose which of these List implementations best suits your code and the desired complexity requirements.

Desired Complexity & Notes

`O(k`^{2}`n)`^{A} time, where `n` is the number of characters generated.
`O(kd)` space, where `k` is the integer input parameter, and `d` is the solution.
Since there are 4 bases, `d ≤ 4`^{k} and thus `log d = O(k)`.

Examples

characters generated by `gen`	`k`	output	`n`
`AAAA`	3	`AAA`	4
`AUAAA`	3	`AAA,AUA,UAA`	5

UGCGAAUAUA	3	AAU,AUA,CGA,GAA,GCG,UAU,UGC	10
UUUUCCCCAAAA	12	UUUUCCCCAAAA	12
(no characters)	> 0	0	0

Testing & Autograder

There are limited local tests in `GenesOrder.main` and `tests/GenesOrderTest.java`; see Lab 1 for testing instructions. Submit `GenesOrder.java` to gradescope; see Lab 1 for submission instructions.

GenesCount

Method Signature

```
public static AbstractList<String> genesCount(InputGenerator<Character>
gen, int k)
```

Method Behaviour & Notes

Returns a list of the different genes (substrands) of length `k > 0` in the sequence generated by a given `InputGenerator`, in order from most frequent to least frequent. When multiple genes have the same frequency, order them alphabetically.

Desired Complexity & Notes

`O(k(n+d log d))`^{AE} = `O(k(n+kd))`^{AE} time.
`O(kd)` space.
Note: `n` is the number of characters generated, `k` is the parameter, and `d` is the solution. Since there are 4 bases, `d` $\leq$ 4^k and thus `log d = O(k)`.

Examples

characters generated by `gen`	`k`	output (`d`)	`n`
AAAA	3	AAA	4
AUAAA	3	AAA,AUA,UAA	5
AUAAUAA	3	AUA,UAA,AAU	5
UGCGAAUAUA	3	AUA,AAU,CGA,GAA,GCG,UAU,UGC	10
UUUUCCCCAAAA	12	UUUUCCCCAAAA	12
(no characters)	> 0		0

Testing & Autograder

There are limited local tests in `GenesCount.main` and `tests/GenesCountTest.java`; see Lab 1 for testing instructions. Submit `GenesCount.java` to gradescope; see Lab 1 for submission instructions.

Local Tests

See the Local Tests section of Lab 1; be sure to replace `comp2402w2411` with `comp2402w2414`

Gradescope Autograder

See the Gradescope Autograder section of Lab 1; be sure to replace Lab 1 with Lab 4 where necessary.

How to Get Help

See the How to Get Help section of Lab 1. It will be updated with any new resources rather than duplicating information here.

Common Errors & Fixes

See the Common Errors & Fixes section of Lab 1. It will be updated with any new errors rather than duplicating information here.

Glossary

There is a glossary at the top of the Problem Solving & Programming Tips document. If you can't find a term listed there, please post publicly on piazza so that everyone can benefit from the answer!

Autograder Runtimes for Sample Solutions

To give you a sense for the autograder runtimes for "perfect scores" you can see some of the screenshots that follow.

addChild

addChild correctness ~5 nodes k=1 (0/0)

```
Running test (1) addChild correctness ~5 nodes k=1
        Test passed.
        Time taken: 0:00:00.890379
```

addChild correctness ~10 nodes long skinny k=2 (0.5/0.5)

```
Running test (2) addChild correctness ~10 nodes long skinny k=2
        Test passed.
        Time taken: 0:00:01.003476
```

addChild correctness ~10 nodes balanced k=2 (0.5/0.5)

```
Running test (3) addChild correctness ~10 nodes balanced k=2
        Test passed.
        Time taken: 0:00:01.011344
```

addChild correctness ~10 nodes random structure k=2 (0.5/0.5)

```
Running test (4) addChild correctness ~10 nodes random structure k=2
        Test passed.
        Time taken: 0:00:00.923837
```

addChild exceptions (0.5/0.5)

```
Running test (5) addChild exceptions
        Test passed.
        Time taken: 0:00:01.034277
```

addChild correctness ~30 nodes, random structure k=3 (0.5/0.5)

```
Running test (6) addChild correctness ~30 nodes, random structure k=3
        Test passed.
        Time taken: 0:00:01.224481
```

addChild O(n) time (0.5/0.5)

```
Running test (7) addChild O(n) time
        Test passed.
        Time taken: 0:00:01.109479
```

addChild O(1) time (4.5/4.5)

```
Running test (8) addChild O(1) time
        Test passed.
        Time taken: 0:00:01.635391
```

addChild O(1) space and O(1) call stack (2.5/2.5)

```
Running test (9) addChild O(1) space and O(1) call stack
        Test passed.
        Time taken: 0:00:02.095690
```

PhyloTree

PhyloTree correctness, some null, k=1 (0.5/0.5)

```
Running test (11) PhyloTree correctness, some null, k=1
        Test passed.
        Time taken: 0:00:00.971501
```

PhyloTree correctness k=2 (0.5/0.5)

```
Running test (12) PhyloTree correctness k=2
        Test passed.
        Time taken: 0:00:01.011076
```

PhyloTree correctness k>=2 (0.5/0.5)

```
Running test (13) PhyloTree correctness k>=2
        Test passed.
        Time taken: 0:00:00.851926
```

PhyloTree exceptions (0.5/0.5)

```
Running test (14) PhyloTree exceptions
        Test passed.
        Time taken: 0:00:00.917263
```

PhyloTree O(n^2) time (0.5/0.5)

```
Running test (15) PhyloTree O(n^2) time
        Test passed.
        Time taken: 0:00:00.871402
```

PhyloTree O(n) time (4.5/4.5)

```
Running test (16) PhyloTree O(n) time
        Test passed.
        Time taken: 0:00:01.617529
```

PhyloTree O(n) space and O(1) call stack (2.5/2.5)

```
Running test (17) PhyloTree O(n) space and O(1) call stack
        Test passed.
        Time taken: 0:00:01.453938
```

LCA

LCA(sib, sib) correctness, k>=2 (0.5/0.5)

```
Running test (20) LCA(sib, sib) correctness, k>=2
        Test passed.
        Time taken: 0:00:00.917021
```

LCA(parent,child) correctness, k>=2 (0.5/0.5)

```
Running test (21) LCA(parent,child) correctness, k>=2
        Test passed.
        Time taken: 0:00:00.986909
```

LCA(s1, s2) for many combinations, k>=2 (0.5/0.5)

```
Running test (22) LCA(s1, s2) for many combinations, k>=2
        Test passed.
        Time taken: 0:00:01.028533
```

LCA exceptions (0.5/0.5)

```
Running test (23) LCA exceptions
        Test passed.
        Time taken: 0:00:00.958338
```

LCA O(n) time (0.5/0.5)

```
Running test (24) LCA O(n) time
        Test passed.
        Time taken: 0:00:01.061863
```

LCA O(1) time (4.5/4.5)

```
Running test (25) LCA O(1) time
        Test passed.
        Time taken: 0:00:01.535447
```

LCA O(n) space and O(1) call stack (2.5/2.5)

```
Running test (26) LCA O(n) space and O(1) call stack
        Test passed.
        Time taken: 0:00:01.526890
```

fixUp

fixUp correctness, many left rotations, k>=2 (0.5/0.5)

```
Running test (31) fixUp correctness, many left rotations, k>=2
        Test passed.
        Time taken: 0:00:00.924202
```

fixUp correctness, both rotations, k>=2 (0.5/0.5)

```
Running test (32) fixUp correctness, both rotations, k>=2
        Test passed.
        Time taken: 0:00:01.074886
```

fixUp exceptions (0.5/0.5)

```
Running test (33) fixUp exceptions
        Test passed.
        Time taken: 0:00:00.901181
```

fixUp O(n) time nodes near root (0.25/0.25)

```
Running test (34) fixUp O(n) time nodes near root
        Test passed.
        Time taken: 0:00:00.944982
```

fixUp O(1) time nodes near root (2.25/2.25)

```
Running test (35) fixUp O(1) time nodes near root
        Test passed.
        Time taken: 0:00:01.629240
```

fixUp O(n) time nodes far from root (0.25/0.25)

```
Running test (36) fixUp O(n) time nodes far from root
        Test passed.
        Time taken: 0:00:00.960360
```

fixUp O(1) time nodes far from root (2.25/2.25)

```
Running test (37) fixUp O(1) time nodes far from root
        Test passed.
        Time taken: 0:00:01.605332
```

fixUp O(1) space and O(1) call stack nodes near root (1.25/1.25)

```
Running test (38) fixUp O(1) space and O(1) call stack nodes near root
        Test passed.
        Time taken: 0:00:01.629480
```

fixUp O(1) space and O(1) call stack nodes far from root (1.25/1.25)

```
Running test (39) fixUp O(1) space and O(1) call stack nodes far from root
        Test passed.
        Time taken: 0:00:01.503864
```

computeSets

computeSets correctness all leaves same (0.25/0.25)

```
Running test (40) computeSets correctness all leaves same
        Test passed.
        Time taken: 0:00:01.245633
```

computeSets correctness all sets disjoint (0.25/0.25)

```
Running test (41) computeSets correctness all sets disjoint
        Test passed.
        Time taken: 0:00:01.106720
```

computeSets correctness sets partially overlap (0.5/0.5)

```
Running test (42) computeSets correctness sets partially overlap
        Test passed.
        Time taken: 0:00:01.217405
```

computeSets correctness all leaves same random structure (0.5/0.5)

```
Running test (43) computeSets correctness all leaves same random structure
        Test passed.
        Time taken: 0:00:01.263231
```

computeSets correctness random assignment (0.5/0.5)

```
Running test (44) computeSets correctness random assignment
        Test passed.
        Time taken: 0:00:01.263555
```

computeSets exceptions (0.5/0.5)

```
Running test (45) computeSets exceptions
        Test passed.
        Time taken: 0:00:00.981536
```

computeSets O(n^2) time (0.5/0.5)

```
Running test (46) computeSets O(n^2) time
        Test passed.
        Time taken: 0:00:01.188615
```

computeSets O(n) time (4.5/4.5)

```
Running test (47) computeSets O(n) time
        Test passed.
        Time taken: 0:00:02.246903
```

computeSets O(1) space and O(1) call stack (2.5/2.5)

```
Running test (48) computeSets O(1) space and O(1) call stack
        Test passed.
        Time taken: 0:00:02.836965
```

Genes

Genes correctness edge case empty strand (0.5/0.5)

```
Running test (4) Genes correctness edge case empty strand
        Test passed.
        Time taken: 0:00:00.774596
```

Genes correctness random (1/1)

```
Running test (5) Genes correctness random
        Test passed.
        Time taken: 0:00:00.755454
```

Genes O(n^2) time n~5,000, k~O(n), d~O(n) (0/0)

```
Running test (6) Genes O(n^2) time n~5,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:00.841082
```

Genes O(nk^2) time n~100,000, k~100, d~O(n) (0.5/0.5)

```
Running test (7) Genes O(nk^2) time n~100,000, k~100, d~O(n)
        Test passed.
        Time taken: 0:00:01.145054
```

Genes O(nk) time n~15,000, k~O(n), d~O(n) (0.5/0.5)

```
Running test (8) Genes O(nk) time n~15,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:00.846708
```

Genes O(n log d) time n~400,000, k~O(n), d~O(n) (2/2)

```
Running test (9) Genes O(n log d) time n~400,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:01.627077
```

Genes O(n) time n~2,000,000, k~O(1), d~O(n) (2/2)

```
Running test (10) Genes O(n) time n~2,000,000, k~O(1), d~O(n)
        Test passed.
        Time taken: 0:00:02.152718
```

Genes O(n) space n~10,000, k, d=O(n) (0.5/0.5)

```
Running test (11) Genes O(n) space n~10,000, k, d=O(n)
        Test passed.
        Time taken: 0:00:00.891074
```

Genes O(k+d) space n=1,000,000, k, d=O(n) (2/2)

```
Running test (12) Genes O(k+d) space n=1,000,000, k, d=O(n)
        Test passed.
        Time taken: 0:00:03.287951
```

GenesOrder

GenesOrder correctness structured random (1/1)

```
Running test (3) GenesOrder correctness structured random
        Test passed.
        Time taken: 0:00:00.777766
```

GenesOrder correctness edge case empty strand (0.5/0.5)

```
Running test (4) GenesOrder correctness edge case empty strand
        Test passed.
        Time taken: 0:00:00.713601
```

GenesOrder correctness random (1/1)

```
Running test (5) GenesOrder correctness random
        Test passed.
        Time taken: 0:00:00.943812
```

GenesOrder O(k n^2) time n~5,000, k~O(n), d~O(1) (0/0)

```
Running test (6) GenesOrder O(k n^2) time n~5,000, k~O(n), d~O(1)
        Test passed.
        Time taken: 0:00:00.927420
```

GenesOrder O(k^2 n) time n~100,000, k~1,000, d~O(1) (1/1)

```
Running test (7) GenesOrder O(k^2 n) time n~100,000, k~1,000, d~O(1
        Test passed.
        Time taken: 0:00:01.544293
```

GenesOrder O(k^2 d) time n~15,000, k~O(n), d~O(n) (2/2)

```
Running test (8) GenesOrder O(k^2 d) time n~15,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:01.997045
```

GenesOrder O(k^2 d) time n~20,000, k~O(n), d~O(n) (2/2)

```
Running test (9) GenesOrder O(k^2 d) time n~20,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:01.734473
```

GenesOrder O(n) space n=10,000, k, d=O(1) (0.5/0.5)

```
Running test (10) GenesOrder O(n) space n=10,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:00.833136
```

GenesOrder O(kd) space n=1,000,000, k, d=O(1) (2/2)

```
Running test (11) GenesOrder O(kd) space n=1,000,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:01.784532
```

GenesCount

GenesCount correctness random (1/1)

```
Running test (5) GenesCount correctness random
        Test passed.
        Time taken: 0:00:00.982121
```

GenesCount O(k n^2) time n~5,000, k~O(n), d~O(1) (0/0)

```
Running test (6) GenesCount O(k n^2) time n~5,000, k~O(n), d~O(1)
        Test passed.
        Time taken: 0:00:00.953249
```

GenesCount O(k^2 n) time n~60,000, k~1,000, d~O(1) (1/1)

```
Running test (7) GenesCount O(k^2 n) time n~60,000, k~1,000, d~O(1)
        Test passed.
        Time taken: 0:00:01.813724
```

GenesCount O(k^2 d) time n~8,000, k~O(n), d~O(n) (2/2)

```
Running test (8) GenesCount O(k^2 d) time n~8,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:02.061771
```

GenesCount O(k^2 d) time n~10,000, k~O(n), d~O(n) (2/2)

```
Running test (9) GenesCount O(k^2 d) time n~10,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:01.761667
```

GenesCount O(n) space n=10,000, k, d=O(1) (0.5/0.5)

```
Running test (10) GenesCount O(n) space n=10,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:00.745329
```

GenesCount O(kd) space n=800,000, k, d=O(1) (2/2)

```
Running test (11) GenesCount O(kd) space n=800,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:01.163159
```