Programming Assignment 4 (Tree Traversals, Sets & Maps, Topological Sorting, and Breadth First Search)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

1 Overview

We are going to:

- **Implement the three types of tree traversals**.
- **Learn how to use ordered sets and maps**

  These are essentially balanced binary search trees (such as AVL tree). In C++, these are called `set` and `map`. In Java, these are called `TreeSet` and `TreeMap`. These are ordered because the values are logically stored and can be retrieved in a sorted order.
- **Learn how to use unordered sets and maps**

  These are essentially hash tables. In C++, these are called `unordered_set` and `unordered_map`. In Java, these are called `HashSet` and `HashMap`.
- **Use Topological Sorting to help mutants**.
- **Implement Breadth First Search, and use it to count the number of components in a graphs and also compute the transitive closure**.

To this end, **your task is to implement the following methods**:

- `preOrder`, `inOrder`, and `postOrder` in `TreeTraversals.h`/`TreeTraversals.java`
- `sortedAlphabet`, `bstSort`, `zeroSumSubArray`, and `kHeavyHitters` in `SetsAndMaps.h`/`SetsAndMaps.java`
- `readLanguage`, `makeGraph`, and `getOrder` in `MutantLanguage.h`/`MutantLanguage.java`
- `initialize`, `traverse`, `execute`, `computeTransitiveClosure`, and `countComponents` in `BFS.h`/`BFS.java`

The project also contains additional files (which you do not need to modify).
Use `TestCorrectness.cpp`/`TestCorrectness.java` to test your code.
For each part, you will get an output that you can match with the output I have given to verify whether your code is correct, or not. Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

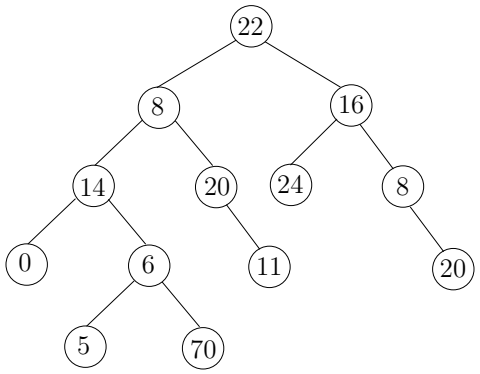
2 Tree Traversals

See the assignment folder for explanation videos about the 3 types of tree traversals: `pre-Order`, `in-Order`, and `post-Order`. Use the pseudo-code for `pre-Order` traversal below and the logic for the other 2 traversals to implement all 3 methods. You MUST use recursion.

Pre-Order Traversal

- print the value in the node, followed by a space (use a space and not a newline to get the desired format).
- if the node has a left child, then recursively carry out a pre-order traversal starting from node's left child
- if the node has a right child, then recursively carry out a pre-order traversal starting from node's right child

The following binary tree has been used to test the three methods.



3 C++ and Java Dynamic Arrays

Here, you will use their C++ and Java implementations of dynamic arrays, which are named respectively **vector** and **ArrayList**. This involves the use of generics/templates. I will discuss this in the context of integers, but you will be using vectors/array-lists of Edge-type linked-lists; you will find enough details in this description pdf for this usage.

- In C++, to create an integer vector use: **vector⟨int⟩ name;**

  To add a number (say 15) at the end of the vector, the syntax is **name.push_back(15)**.

  Although not needed for this assignment, the following are good to know. To remove the last number, the syntax is **name.pop_back()**. To access the number at a particular index (say 4), the syntax is **name.at(4)**.

- In Java, to create an integer ArrayList use: **ArrayList⟨Integer⟩ name = new ArrayList⟨Integer⟩();**

  To add a number (say 15) at the end of the array list, the syntax is **name.add(15)**.

  Although not needed for this assignment, the following are good to know. To remove the last number, the syntax is **name.remove(name.size() − 1)**. To access the number at a particular index (say 4), the syntax is **name.get(4)**.

Note that the push_back and add functions are equivalent to *insertAtEnd*, whereas pop_back and remove (with the appropriate index) are equivalent to *deleteLast*.

4 Sets and Maps

A Set is essentially a set in the typical Math terminology – contains unique elements (or more commonly referred to as *keys*). A Map on the other hand consists of map entries, where each map entry comprises of a *key* and *value* pair. You can search the map for a particular key, and if it exists, retrieve the corresponding value back. Thus a map essentially maps an element (called *key*) uniquely to another element (called *value*).

Sets are implemented as a balanced search tree (aka ordered sets) or as a hash-table (aka unordered sets). Maps can be implemented as a balanced search tree (aka ordered maps) or as a hash-table (aka unordered maps). In both cases, keys can be of any type (integers, floats, strings, characters, objects of a class); likewise, values (in case of maps) can also be of any type. For unordered sets and maps, we must be able to check whether or not two keys are the same. For ordered sets and maps, we must be able to order two keys (such as dictionary order for strings).

Operations on an ordered set/map are typically supported in $O(\log n)$ time, where n is the number of items in the set/map. Operations on an unordered set/map are typically supported in $O(1)$ expected (average) time but in the worst-case it may take $O(n)$ time, where n is the number of items in the set/map. For this assignment, we will assume $O(1)$ time for unordered sets/maps as the worst-case rarely happens.

An important thing to consider is that the *keys* that are stored in ordered sets and maps are, as the name suggests, *ordered*, such as in numeric order (for integers and floats), or in lexicographic order (for strings). Therefore, whenever an application demands that keys be stored in some well-defined order, we should consider ordered sets and maps. On the other hand, when the order is not important, then we should use unordered sets and maps because they are faster.

Over here, we will see a few applications of sets and maps. In the project folder you will find videos explanations of how some of these and related applications are implemented via sets and maps. Let's first see how they are implemented in C++ and Java.

4.1 C++

I will list some of the functions of sets and maps (both ordered and unordered). For more details, check out C++'s documentation.

Ordered and Unordered Set

- To create an integer ORDERED set: `set<int> mySet;`

  To create an integer UNORDERED set: `unordered_set<int> mySet;`

  Note that you may need to change the types of keys stored according to the application.

- To get the size of the set: `mySet.size();`

- To insert an integer x: `mySet.insert(x);` this will add x if it is not present, else no change.

- To check if an integer x is already in the set: `if(mySet.find(x) != mySet.end())`. The statement inside the if evaluates to true if and only if x is already in the set.

- To create an iterator on an (ORDERED or UNORDERED) set and print all values:

```
set<int>::iterator it = mySet.begin();
while (it != mySet.end()) {
    cout << *it << " ";
    ++it;
}
```

For unordered set, just create `unordered_set<int>::iterator it`; rest is the same.

Ordered and Unordered Map

- To create an ORDERED map with integer keys and character values: `map<int, char> myMap;`

  To create an UNORDERED map with integer keys and character values: `unordered_map<int, char> myMap;`

  Note that you may need to change the types of keys and values according to the application.

- To get the size of the map: `myMap.size();`

- To insert an integer *key-value* pair: `myMap[key] = value;` this will add the pair if it is not present, else it will update the existing value of *key* with the "new" value.

- To check if the map already contains a particular *key*: `if(myMap.find(key) != myMap.end())`

  The statement inside the if evaluates to true if and only if *key* is already in the map.

- To retrieve the value corresponding to a particular *key*: `int value = myMap[key];`

  If *key* is not present, since this method won't typically return a NULL, to avoid unexpected results, you should first check if *key* is present before using this.

- To create an iterator on an (ORDERED or UNORDERED) map and print all keys/values:

```
// create an iterator on the set of keys stored in the map
map<int, char>::iterator it = myMap.begin();
while (it != myMap.end()) { // as long as there is an entry
    int key  = it -> first; // obtain the key from the iterator
    char value = myMap[key]; // get the value corresponding to the key
                             // can also use: char value = it -> second;
    ++it; // move the iterator to the next entry (if exists)
    cout << key << ": " << value << endl;
}
```

For unordered map, just create `unordered_map<int, char>::iterator it`; rest is the same.

4.2 Java

I will list some of the functions of sets and maps (both ordered and unordered). For more details, check out Java's documentation.

Ordered and Unordered Set

- To create an integer ORDERED set: `TreeSet<Integer> mySet = new TreeSet<>();`

  To create an integer UNORDERED set: `HashSet<Integer> mySet = new HashSet<>();`

  Note that you may need to change the types of keys stored according to the application.

- To get the size of the set: `mySet.size();`

- To insert an integer x: `mySet.add(x);` this will add x if it is not present, else no change.

- To check if an integer x is already in the set: `if(mySet.contains(x))`. The statement inside the if evaluates to true if and only if x is already in the set.

- To create an iterator on a set (ORDERED or UNORDERED) and print all values:

```
Iterator<Integer> it = mySet.iterator();
while (it.hasNext())
    System.out.print(it.next() + " ");
```

Ordered and Unordered Map

- To create an ORDERED map with integer keys and character values: `TreeMap<Integer, Character> myMap = new TreeMap<>();`

  To create an UNORDERED map with integer keys and character values: `HashMap<Integer, Character> myMap = new HashMap<>();`

  Note that you may need to change the types of keys and values according to the application.

- To get the size of the map: `myMap.size();`

- To insert an integer *key-value* pair: `myMap.put(key, value);` this will add the pair if it is not present, else it will update the existing value of *key* with the "new" value.

- To check if the map already contains a particular *key*: `if(myMap.containsKey(key))`

  The statement inside the if evaluates to true if and only if *key* is already in the map.

- To retrieve the value corresponding to a particular *key*: `Integer value = myMap.get(key);`

  The method returns *null* if *key* is not present.

- To create an iterator on a map (ORDERED or UNORDERED) and print all keys/values:

```
// create an iterator on the set of keys stored in the map
Iterator<Integer> it = myMap.keySet().iterator();
while (it.hasNext()) { // as long as there is a key
    Integer key = it.next(); // obtain the key from the iterator,
                             // and move iterator to the next key (if exists)
    Character value = myMap.get(key); // get the value corresponding to the key
    System.out.println(key+ ": " + value);
}
```

4.2.1 Alphabet Finder using Ordered Set

The first method simply returns the alphabet of a given array of characters in sorted order, i.e., this method returns the distinct characters in the array in sorted order. Note that since we want things in sorted order and all we really care about are the characters (with nothing else tied to them), we should lean towards an ordered set. Fill up the `sortedAlphabet` method:

Alphabet Finder

- Note that the ordered set will directly solve this. So, create an ORDERED set. You will also need to return the alphabet in a dynamic array; so, create one.

  Notice that the input array contains characters; so, your dynamic array and ordered set should also be able to store characters.

- Now, take all the characters from the array and insert them into the ordered set.
- Since sets will only keep unique characters, you are pretty much done – all duplicates have been removed. Just an use iterator on the set to retrieve the numbers one-by-one, and insert them into a dynamic array.
- Once all numbers have been added, return the dynamic array.

Your code must have complexity $O(n \log \sigma)$, where n is the length of the input array and σ is the alphabet size. Otherwise, you will get partial credit, even if you get the correct output.

4.2.2 Sorting using Ordered Map

The second method sorts an array of characters. Notice that a character can appear multiple times; therefore, in the sorted order, each character must appear as many time as present in the original array. Therefore, we will need an ordered map (ordered because we need the output to be sorted, and a map because we will store the number of occurrences of each character as value).

To sort using an ordered map, we will essentially employ the following idea. We compute the frequency (*value*) of each character (*key*) in the input array. For example, if the array is $[b, a, n, a, n, a]$, the we compute frequencies as $[a : 3, b : 1, n : 2]$. To sort the input array: simply write 3 a's, followed by 1 b, followed by 2 n's. Note that the use of the ordered map will ensure that we get a, followed by b, followed by n. Use this idea and fill up the `bstSort` method:

Sorting Using a Map

- First create a character-integer ORDERED map.
- Now, loop over the array. Within the loop:
  - If the current character in the array is not in the map as a key, then insert it into the map as key with value one. This implies that this is the first time you are seeing the character, and so its frequency is one.
  - Else, get the stored frequency (i.e., value) of the current character (as key) from the map. Add one to this value and reinsert into the map (as value once again) with key as the current character. This implies that you have seen the current character in the string prior to this occurrence, and so its frequency should be increased by one.

- At this point, all frequencies have been computed and are stored in the map as a character to frequency map. So, iterate over the map; note that you will iterate in sorted order of keys. For each key c, obtain the corresponding value v; this tells you how many copies of c exists in the original array. So, place v copies of c one by one into the $arr[\,]$ array; note that you may need a counter to go over the array $arr[\,]$

Your code must have complexity $O(n \log \sigma)$, where n is the length of the input array and σ is the alphabet size. Otherwise, you will get partial credit, even if you get the correct output.

4.2.3 Checking if a subarray sums to zero using Unordered Set

The third method checks if an array contains a subarray whose sum is zero. It returns true or false accordingly. Notice that we do not really care about order (just an Yes or No answer is all we desire). So, we will use an unordered set; it may not be obvious right now why a set suffices, but that will become clearer in the due course of the discussion.

Let's understand the idea first. Pick an example array: $[12, -26, 1, 8, 9, -6, 4, -12, -3, 12]$. Note that this array contains a sub-array that sums to zero; specifically, the numbers: $8, 9, -6, 4, -12$ and -3. On the other hand, $[-6, 4, -12, -3, 12]$ does not have a sub-array summing to zero.

One obvious approach is to find sum of all subarrays and see if any one of them is zero, but this will take $O(n^2)$ time. We will see how to solve this in $O(n \log n)$ time.

Let's define $prefixSum(i) = arr[0] + arr[1] + arr[2] + \cdots + arr[i]$, i.e., $prefixSum(i)$ is the sum of all numbers from index 0 to index i. Note that if there is a subarray from x to y such that $A[x] + A[x+1] + \cdots + A[y] = 0$, then

$$\begin{aligned} prefixSum(y) &= A[0] + A[1] + A[2] + \cdots + A[x-1] + A[x] + A[x+1] + \cdots + A[y] \\ &= prefixSum(x-1) + A[x] + A[x+1] + \cdots + A[y] \\ &= prefixSum(x-1) \end{aligned}$$

In the following example, notice that the red-colored prefix sum -13 has occurred twice: once at index 2 and then at index 8. You can check that the subarray shaded in blue from index 3 to index 8 has a sum of zero. This makes sense as a zero-sum subarray won't change the prefix sum.

i	0	1	2	3	4	5	6	7	8	9
arr[i]	12	-26	1	8	9	-6	4	-12	-3	12
prefixSum(i)	12	-14	**-13**	-5	4	-2	2	-10	**-13**	19

On the other hand there is no repeated occurrence of a prefix sum for the second example:

i	0	1	2	3	4
arr[i]	-6	4	-12	3	-12
prefixSum(i)	-6	-2	-14	-11	-23

So, whenever we find a prefix sum that we have seen before, we know there is a sub-array summing to zero. To check for duplicate prefix sums, we use an unordered set.[1]

Use this idea and the following sketchy pseudo-code to fill up the `zeroSumSubArray` method:

[1] We are not interested in where the subarray lies; all we need is an Yes or No answer. If you care to find the indexes which make up the subarray, use an ordered set. Notice that this subarray starts at the index immediately after last occurrence of the prefix sum and ends at the current index.

Zero-Sum Subarray Checker

- Create an integer UNORDERED set.
- Initialize $prefixSum$ to 0
- Traverse through the array and do the following:
  - Update $prefixSum$ by adding the current number in the array to it
  - If $prefixSum$ is zero, then there is obviously a zero-sum array. So, return $true$.
  - Now, check if the set contains the prefix sum. If it does, then we have seen this prefix sum before. That means we must have a zero-sum array. So, return $true$.
  - A match with prefix sum was not found. So, insert the prefix sum into the set.
- You never found a zero-sum sub-array; so, return $false$.

Your code must have complexity $O(n)$, where n is the length of the input array. Anything worse, you will get partial credit, even if you get the correct output.

4.3 k-Heavy Hitters

Assume that there are n numbers; given a k, the problem is find the numbers that occur more than n/k times. For example, if the numbers are $[12, 13, 12, 15, 13, 12, 15, 12, 13, 12, 13, 12]$ with $n = 12$. The number 12 occurs 6 times, 13 occurs 4 times, and 15 appears twice. Then, there is no 2-heavy hitter (also called a majority) in the array (because that would mean a number has to occur more than $12/2 = 6$ times). The only 3-heavy hitter is the number 12. The 4-, 5-, and 6-heavy hitters are 12 and 13. Every number is a 7-heavy hitter.

The k-heavy hitters problem is one of the most important problems related to social media. For example, Facebook uses a fast memory (called cache) to store profile pages that you visit most frequently, which allows faster retrieval of cached pages. This fast memory is pretty expensive (and by extension small), and so one must be judicious in determining which pages to store. One way is to use the heavy hitters problem; say among all the n pages that you visit, Facebook wants to cache the ones that you visit more than $n/10$ times. This is k-heavy hitters with $k = 10$.

k-heavy hitters

- Create an integer UNORDERED map (be careful with the data types for the map).
- Traverse through the array to compute the frequency of each number and store them in the hashmap (same technique as you did before with sorting via map).
- Now, you just report those numbers from the map whose frequency is more than n/k. These frequent numbers are to be added to a dynamic array and returned back.

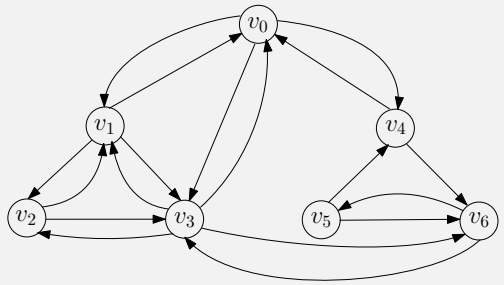
  Notice that there are no duplicates in the output. One way to ensure that is by setting the frequency of a number to zero (in the map) when you have already added that to the dynamic array.

Your code must have complexity $O(n)$, where n is the length of the input array. Anything worse, you will get partial credit, even if you get the correct output.

5 Graph Data

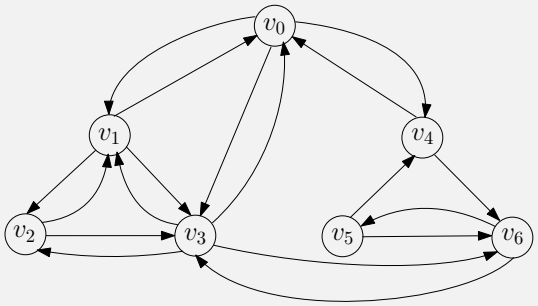
directed1.txt and corresponding graph

```
7
3 1 3 4
3 0 2 3
2 1 3
4 0 1 2 6
2 0 5
2 4 6
2 3 5
```



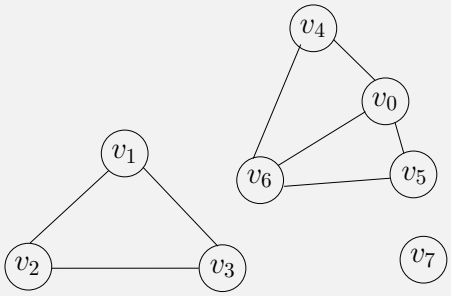
directed1.txt and corresponding graph

```
7
2 1 3
3 0 2 3
2 1 3
4 0 1 2 5
3 0 5 6
1 3
1 4
```



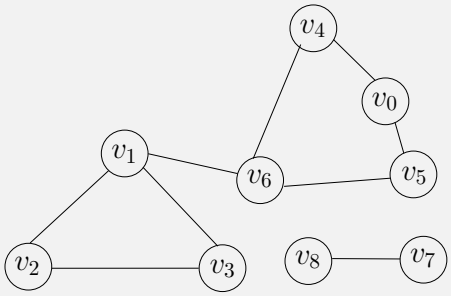
undirected1.txt and corresponding graph

```
8
3 4 5 6
2 2 3
2 1 3
2 1 2
2 0 6
2 0 6
3 0 4 5
0
```



undirected2.txt and corresponding graph

```
9
2 4 5
3 2 3 6
2 1 3
2 1 2
2 0 6
2 0 6
3 1 4 5
1 8
1 7
```



To test BFS, we use the graphs in files: `directed1.txt,` and `directed2.txt`. To test ComponentCounter, we use the graphs in files: `undirected1.txt,` and `undirected2.txt`. To test TransitiveClosure, we use all four graphs. Next, a description is given.

Consider the first graph; the file `directed1.txt` represents this graph. The first line in the file stores the number of vertices, which is 7. Then, there are 7 lines, representing v_0 through v_6. The first number in each line is the number of outgoing edges of that vertex. Then, we have the vertices to which each outgoing edge leads. For e.g., in line 2 (i.e., 3 1 3 4): the first number is 3, indicating vertex v_0 has 3 outgoing edges to the vertices v_1, v_3, and v_4 respectively.

5.1 Adjacency List: Representing Graphs in Memory

The vertices in the graph are numbered 0 through $n-1$, where n is the number of vertices. We use **adjList** (called *adjacency list*) to represent the graph; this is conceptuall similar to a two-dimensional jagged array, with the difference that each row is implemented using a linked list. Specifically, row index i in the array corresponds to the vertex v_i, i.e., row 0 corresponds to v_0, row 1 corresponds to v_1, and so on.

Each row is a linked list of type `Edge`, and each node in the linked list for row i stores an outgoing edge of the vertex v_i. Each edge has 2 properties – *src*, and *dest*, which are respectively the vertex from which the edge originates, and the vertex where the edge leads to.

As an example, consider the first graph in the previous page. Row 0 of adjList contains the following edges: $\langle 0,1\rangle$, $\langle 0,3\rangle$, and $\langle 0,4\rangle$; this is to be interpreted as vertex v_0 has 3 outgoing edges – one edge to the vertex v_1, one to the vertex v_3, and another to vertex v_4. Likewise, row 1 contains the edges $\langle 1,0\rangle$, $\langle 1,2\rangle$, and $\langle 1,3\rangle$. Likewise, for the other vertices/rows.

In a nutshell, **Edge is a class which has two integer variables** – *src* **and** *dest*. The `adjList` structure in the `Graph` class, therefore, is a dynamic array of linked lists, where each node in the linked-list is of type Edge. In C++, we implement *adjList* as a vector of Edge linked-lists. In Java, we implement *adjList* as an array-list of Edge linked-lists.

In the project folder, you will find an explanation video on how adjacency lists are implemented. The only difference in this implementation is that you are using linked lists for each row, instead of (dynamic) arrays. See the `Graph.h/Graph.java` files for the structure of the graph (i.e., how it is stored in the memory). Also, see `UnderstandingAdjacencyList.cpp/UnderstandingAdjacencyList.java` files for how an adjacency list (as formulated in `Graph.h/Graph.java`) works for a particular graph (the first one in the previous page).

6 C++ and Java In-built Linked List

We will use the inbuilt linked-list of C++ or Java. This is an implementation of a doubly-linked-list (with links going both forward and backward). It supports all the standard operations (inserting at front or end, deleting head or tail, traversing the list, etc.)

I'll highlight some of the usages (not all may be required). Typically, this encompasses use of generics (in Java) or templates (in C++), whereby you can create linked lists of any type (and not just integer). I will discuss integer linked lists, but lists of any type can be created; in fact, you will be using linked lists of type `Edge`. The syntax will remain pretty much the same, and this description file contains enough details on how to adapt for Edge type linked lists.

6.1 C++

- Syntax to create an integer list: `list<int> nameOfList;`

- To get the size of the list, the syntax is: `nameOfList.size();`
- To add a number at the end, the syntax is: `nameOfList.push_back(15);` To add a number at the beginning, the syntax is: `nameOfList.push_front(15);`
- To remove the last number, the syntax is: `nameOfList.pop_back();` To remove the first number, the syntax is: `nameOfList.pop_front();`

To traverse the list (say for retrieving a value or searching or printing or deleting), one can use an iterator as shown next. Here, `print` function prints the list.

```
static void print(list<int> &numbers) { // printing the content of the list
    list<int>::iterator it = numbers.begin();
    while (it != numbers.end()) {
        cout << *it << " ";
        it++;
    }
    cout << endl;
}
```

- `list<int>::iterator it = numbers->begin()` declares an iterator *it* on the list *numbers* and the iterator points to the first number on the list.
- `it != numbers.end()` ensures that the iterator traverses until the end of the list is reached.
- `*it` retrieves the value of the node at which the iterator is pointing, and thus `cout << *it` prints the value of the node at which the iterator is pointing.
- `it++` moves the iterator to the next node.

6.2 Java

- Syntax to create an integer list: `LinkedList<Integer> nameOfList = new LinkedList<Integer>();`
- To get the size of the list, the syntax is: `nameOfList.size();`
- To add a number at the end, the syntax is: `nameOfList.addLast(15);` To add a number at the beginning, the syntax is: `nameOfList.addFirst(15);`
- To remove the last number, the syntax is: `nameOfList.removeLast();` To remove the first number, the syntax is: `nameOfList.removeFirst();`

To traverse the list (say for retrieving a value or searching or printing or deleting), one can use an iterator as shown next. Here, `print` method prints the list.

```
static void print(LinkedList<Integer> numbers) { // printing the content
    Iterator<Integer> it = numbers.iterator();
    while (it.hasNext())
        System.out.print(it.next() + " ");
}
```

- `Iterator<Integer> it = numbers.iterator();` declares an iterator *it* on the list *numbers* and the iterator points to the first number on the list.
- `while (it.hasNext())` ensures that the iterator traverses until the end of the list is reached.
- `it.next()` retrieves the value of the node at which the iterator is pointing as well as moves the iterator to the next node. Thus `System.out.print(it.next())` prints the value of the node at which the iterator is pointing, and then moves the iterator to the next node.

7 Implementing Queue with a Linked List

7.1 C++

- To create a queue: `list<int> name`;
- To find the size of the queue: `name.size()`;
- To enqueue a number onto the queue: `name.push_back(15)`;
- To dequeue a number from the queue: use `name.front();` to get the first number. Then use `name.pop_front();` to remove it.

7.2 Java

- To create a queue: `List<Integer> name = new LinkedList<>()`;
- To find the size of the queue: `name.size();`
- To enqueue a number onto the queue: `name.addLast(15);`
- To dequeue a number from the queue: `name.removeFirst();`

8 Mutant Language

Professor Xavier is trying to develop a new language for fellow mutants. As with any language, the main constituent are words via which one can effectively communicate with one another. However, the language is new, and mutants are not yet conversant about the meaning of the words; therefore, professor Xavier has to design a standard dictionary which will map mutant words to their English meaning. Since fellow mutants need to be able to search the dictionary, the Professor needs to make sure that words can be ordered (like a normal English dictionary). Therefore, after creating the words, Professor Xavier needs to ensure that there are no cyclic dependencies. Being sympathetic to the mutant cause, we want to help Professor to make sure that the language is meaningful.

8.1 File Description

To test, we use the language in files: `mutant1.txt, mutant2.txt` and `mutant3.txt`. The first line contains two numbers. The first is the number of distinct characters in the language. For example, in `mutant1.txt`, it is 5 (the distinct characters being $\{a, b, c, d, e\}$). The second is the number of words in the language; here, it is 9. Then, we have the 9 words of the language.

8.2 Comparing Strings

Recall how you compare strings lexicographically (i.e., dictionary order wise). You match the strings one character at a time, until you find a mismatch (or exhaust one of the strings). The

lexicographically smaller string is the one with the smaller mismatched character (or the shorter one if you exhaust one string).

8.3 Assumptions

For the sake of simplicity, we assume that the words contains letter from the lower-case English alphabet in a contiguous way. Specifically, it contains the letters starting from a and we will not skip a letter. We will also assume that all words are distinct. Also, Professor Xavier makes sure that if there are two words X and Y, where X is a prefix of Y (such as $X = abc$ and $Y = abca$), she always places X before Y. If that were not the case, more complications arise.

8.4 What we need to ensure?

We have to make sure that the words in the language can be ordered in a dictionary. In other words, the first word must be lexicographically smaller than the second, the second smaller than the third, and so on. However, since this is a new language, it is no longer necessary that a is smaller than b, and b is smaller than c, and so on. We are perfectly happy as long as we can somehow order them (such as $b < a < c < e < d$), and we do not create a cycle (such as $b < a$, $a < c$, and $c < b$).

8.5 How do we do it?

To verify the correctness of the language, we create a graph as follows. For each distinct letter, you create a vertex. Specifically, a corresponds to the vertex v_0, b corresponds to the vertex v_1, and so on. We scan each word one-by-one and compare it to the next one character at a time. Suppose, the first character where the current word mismatches the next one are α and β respectively. Then, we draw an edge from the vertex corresponding to α to the vertex corresponding to β. In `mutant1.txt`, we compare *baa* and *abc* first. Since $b \neq a$, we draw an edge from v_1 to v_0. Now, we compare *abc* and *abca*; since no mismatch occurs (before we exhaust a word), we do not create an edge. Then, we compare *abca* and *cabe*; since $a \neq c$, we draw an edge from v_0 to v_2. Next we compare *cabe* and *cad*; the first two characters match, but then $b \neq d$, and we draw an edge from v_1 to v_3. We proceed like this until we exhaust all the words. See next page for an illustration.

Now, the the language is valid (i.e., we can create a dictionary with the words) if the graph formed is acyclic; in that case, the precedence order of the characters is given by a topological order of the graph. To detect whether or not the graph is acyclic, we apply topological sorting.

8.6 Why does it work?

Observe that whenever we find a mismatch, we draw an edge from the mismatched character of current word to that of next word. Hence, if the graph contains a cycle, then it must mean that there are words $X_1, X_2, \ldots, X_k$ in the language where X_1 is smaller than X_2, X_2 is smaller than X_3, and so on until X_k is smaller than X_1 (because the first mismatched character of X_1 is smaller than X_2, that of X_2 is smaller than X_3, and so on until that of X_k is smaller than X_1). If the graph does not contain a cycle, it must mean that we can order the words lexicographically.

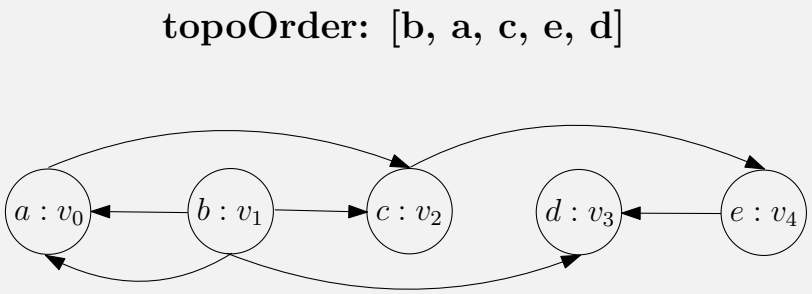
8.7 Examples

In `mutant1.txt`, the graph is acyclic and we have a valid topological order. On the other hand, in `mutant2.txt`, if you choose the words *abcd*, *cab*, *daa*, and *abb*, you realize that there is no way to order these words in that order because that implies $a < c < d < a$. If you look at the corresponding

graph, it has the cycle $a \to c \to d \to a$; there are other cycles as well such as $a \to b \to a$. As long as there is one cycle, the language is not valid.

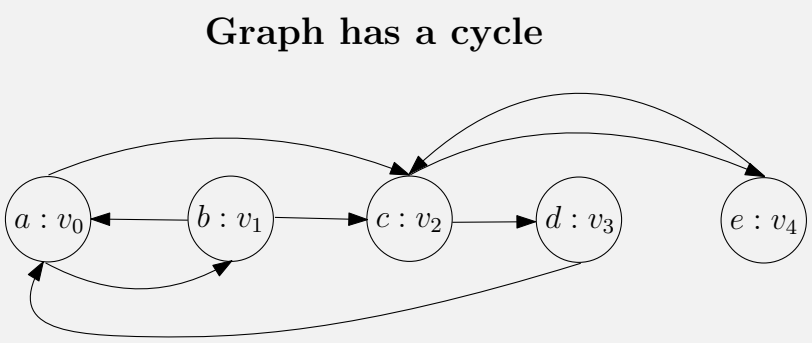
mutant1.txt and corresponding graph

```
5 9
baa
abc
abca
cabe
cad
eba
ecada
dba
daae
```



mutant2.txt and corresponding graph

```
5 9
baa
abcd
cab
eba
eca
cba
daa
abb
bcd
```



8.8 Using ASCII to map character to vertex

One thing to note is that we have to map lower-case English characters $(a, b, c, \ldots,)$ to vertex numbers $(0, 1, , 2, \ldots,)$. This is because our Graph structure uses integers as vertices (which start at zero). What we essentially do is use the ASCII value of the character and subtract 97 to map it to its rank among the lower-case letters of the English alphabet. The ASCII value of a is 97, b is 98, and so on; see http://www.asciitable.com/ for the full ASCII table. Therefore, if we have the characters a, b, c, d, and e, then they are mapped respectively to $0, 1, 2, 3$, and 4 (corresponding to the vertices v_0, v_1, v_2, v_3, and v_4).

Likewise, once we get the vertex order (integer values), we transform them to the actual characters by adding 97 to get the ASCII value and typecasting to char.

8.9 One final thing before we move to code

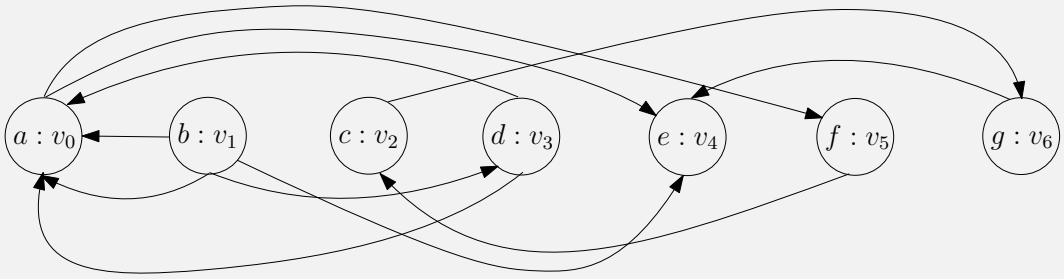
One thing you may have observed is that we can add multiple edges from one vertex to another vertex. For example in `mutant3.txt`, we add two edges from b to a. This is because the words *dbbb* and *dbac* add one edge, and the words *abcd* and *aade* add another edge. There is no reason to add both the edges, but trying to avoid that would mean that we scan the outgoing edges of b to make sure an edge to a is not already added (or use specialized structures such as a hashtable to maintain the outgoing edges). Having multiple edges, however, does not cause any harm (in the

sense that it does not induce any new cycle) and both edges will be relaxed when we process the vertex b. So, once again, in the interest of keeping things simple, we add multiple edges.

mutant3.txt and corresponding graph

```
7 14
baa
dbbb
dbac
dbacd
abcd
aade
faab
feabc
cab
caef
gdfg
gagfr
ea
eae
```

topoOrder: [b, d, a, f, c, g, e]



8.10 Pseudo-code

First, implement the `readLanguage` method. You'll find similar syntax in `readUnweightedGraph` and `readWeightedGraph` methods in `Graph.h/Graph.java` file.

Reading mutant file

- **C++:** Create an input file stream $fileReader$ on $filePath$. To create an input file stream on a file `x.txt`: **ifstream fileReader("x.txt", ios::in);**

  Java: Create a Scanner object $fileReader$ on $filePath$. To create a scanner on a file `x.txt`: **Scanner fileReader = new Scanner(new FileInputStream("x.txt"));**

- Read the number of distinct characters into the class-variable $numVertices$.

  C++: Syntax to read a value into an integer variable x is: **fileReader ≫ x;**

  Java: Syntax to read a value into an integer variable x is: **x = fileReader.nextInt();**

- Now, read the number of words in the language into the class-variable $numWords$

- Allocate $numWords$ cells for the class-array $words$. Note that $words$ is a string array.

- Run a for loop from $i = 0$ to $i < numWords$. Within the for-loop, use $fileReader$ to read the line into the array cell $words[i]$.

  C++: Syntax to read into a string variable y is: **fileReader ≫ y;**

  Java: Syntax to read into a string variable y is: **y = fileReader.next();**

- After the loop, close $fileReader$.

Notice that the first number on the file is read as the number of vertices in the graph, as desired.

Now, we will create the graph based on the idea discussed previously. A detailed pseudo-code is given below to achieve the same. Use it to implement the `makeGraph()` method. You'll find similar syntax in the `readUnweightedGraph` method in `Graph.h/Graph.java` file.

Creating the graph

- Allocate $numVertices$ cells for $inDegree$ array.
- Allocate $numVertices$ rows for $adjList$.

  C++ syntax: adjList.reserve(numVertices);

  Java syntax: adjList = new ArrayList<>(numVertices);
- for ($i = 0$ to $i < numVertices$), do the following:
  - set $inDegree[\,]$ to 0
  - add a blank row to $adjList$

    C++ syntax: adjList.push_back(list<Edge>());

    Java syntax: adjList.add(new LinkedList<Edge>());

    Adding this blank row is necessary so that we have reserved a row for each vertex, which will save us from a null pointer exception if we try to access an empty row (i.e., a vertex with no outgoing edges) later on.
- for ($i = 0$ to $i < numWords - 1$), do the following:
  - let $currentWord$ be the word at index i of $words[\,]$ array
  - let $nextWord$ be the word at index $i + 1$ of $words[\,]$ array
  - let $minLength$ be the minimum of the lengths of $currentWord$ and $nextWord$. That is, if the lengths of $currentWord$ and $nextWord$ are 7 and 5 respectively, then $minLength = 5$, and vice-versa. Ties are broken arbitrarily.

    C++/Java syntax for length of a string named *str*: str.length()
  - Now, we want to find the first character where $currentWord$ and $nextWord$ mismatch. Let this characters be x and y respectively. We want to draw an edge from the vertex for x to the vertex for y. Since x and y are lower-case characters with ASCII values starting from 97, and the graph can only support integers from 0, we subtract 97 from both x and y to reduce the scale (for any lower-case character) so that it starts from zero.

    for ($j = 0$ to $j < minLength$), do the following:
    - let x be the j^{th} character of $currentWord$

      C++ syntax: char x = currentWord.at(j);

      Java syntax: char x = currentWord.charAt(j);
    - let y be the j^{th} character of $nextWord$
    - if ($x \neq y$), then:
      - let **int** $src = x - 97$
      - let **int** $dest = y - 97$
      - create an edge e by calling the Edge constructor with arguments src and $dest$ respectively

· add the edge e to the end of $adjList[src]$
C++ syntax: adjList.at(src).push_back(e);
Java syntax: adjList.get(src).add(e);
· increment $inDegree[dest]$ by one
· break

Finally, we use topological sorting to find the order of the characters or detect that the graph has a cycle. A detailed pseudo-code is given in the next page to achieve the same. Use it to implement the `getOrder()` method in `MutantLanguage.h`/`MutantLanguage.java`.

Finding character order

- Create a char array $topoOrder[\ ]$ having length $numVertices$.

  C++ programmers must use dynamic allocation. So, if you want to return a char array x of length 10, it must be declared as **char *x = new char[10];**

- Create an integer queue $vertexQ$.
- Initialize an integer variable $topoLevel = 0$
- for ($i = 0$ to $i < numVertices$)
  - if ($inDegree[i]$ equals 0), enqueue i
- while ($vertexQ's\ size > 0$), do the following:
  - let v be the vertex obtained by dequeue-ing $vertexQ$
  - assign $topoOrder[topoLevel] =$ (char) $(v + 97)$
  - increment $topoLevel$ by 1
  - let row be the linked list corresponding to the row $adjList[v]$; notice that row is a linked list of type $Edge$
    C++ syntax: $list\langle Edge\rangle\ \&row = adjList.at(v)$;
    Java syntax: $LinkedList\langle Edge\rangle\ row = adjList.get(v)$;
  - declare an iterator on the linked list row
  - while (iterator has not reached the end of the linked list row), do the following:
    - using the iterator, obtain $adjEdge$, which is the next outgoing edge of v, and then move the iterator.
      C++ syntax: $Edge\ \&adjEdge = *it$; followed by $it++$;
      Java syntax: $Edge\ adjEdge = it.next()$;
    - let w be the destination of $adjEdge$
    - decrement $inDegree[w]$ by 1
    - if ($inDegree[w]$ equals 0), then enqueue w to $vertexQ$
- if ($topoLevel \neq numVertices$), return $null$, else return $topoOrder$

  At the end we return $null$ when all vertices have not been added to the topological order, which implies that the graph has a cycle and an ordering of the characters (and hence words) cannot be obtained.

9 Breadth First Search

BFS Initialize

- Allocate $numVertices$ cells for the $level[\,]$ array.
- Use a loop to initialize all cells of $level[\,]$ to ∞.
  In C++, use INT_MAX for ∞. In Java, use Integer.MAX_VALUE for ∞.

BFS Traverse

- Create an integer queue $vertexQ$.
- Enqueue s to $vertexQ$.
- Set $level[s] = 0$.
- while ($vertexQ's\ size > 0$), do the following:
  - let v be the vertex obtained by dequeueing $vertexQ$
  - Similar to what you did for the `createOrder` method in MutantLanguage, use an iterator to visit each adjacent edge $adjEdge$ of v, and do the following:
    - let w be the destination of $adjEdge$
    - if ($level[w]\ equals\ \infty$) then
      - set $level[w]$ to $level[v] + 1$
      - enqueue w to $vertexQ$

BFS Execute

- Initialize BFS
- Traverse from the vertex s

9.1 Counting the number of components

A component in an undirected graph is a subset of vertices (and edges) such that there is a path between every pair of vertices. For example, `undirected1` graph has 3 components – $\{v_0, v_4, v_5, v_6\}$; $\{v_1, v_2, v_3\}$; and $\{v7\}$. Likewise, `undirected12` graph has 2 components – $\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$; and $\{v_7, v_8\}$. In the project folder, you will find a video explaining how to count the number of components using Breadth-First Search.

Count Components

- Initialize BFS
- Initialize a counter to 0
- for ($i = 0$ to $i < numVertices$), do the following:
  - if (level value of vertex i is infinity) then
    - traverse from vertex i

        * increment *counter* by one
    - return the counter

9.2 Computing the transitive closure

A transitive closure is a matrix M such that $M[i][j]$ is true if there is a path from vertex v_i to vertex v_j, else $M[i][j]$ is false. We can easily use BFS to compute the transitive closure as follows: In the project folder, you will find a video explaining how to obtain transitive closure using Depth-First Search. Using virtually the same ideas, your task is compute the transitive closure using Breadth-First Search. To this end, use the following pseudo-code to complete the `computeTransitiveClosure` method.

Compute Transitive Closure

- Create a two-dimensional boolean matrix M that has $numVertices$ many rows.

  C++ syntax: bool **M = new bool*[numVertices];

  Java syntax: boolean[ ][ ] M = new boolean[numVertices][ ];

- for ($i = 0$ to $i < numVertices$), do the following:
  - allocate $numVertices$ many cells for row i of matrix M.
  - execute BFS from vertex i
  - using a loop for $j = 0, 1, 2, \ldots, numVertices - 1$ (all inclusive), set $M[i][j]$ to *true* if $level[j] \neq \infty$ (which means there is a path from i to j), and $false$ otherwise.
- return the matrix M