Metro State University

ICS 240: Introduction to Data Structures

Spring 2025

Programming Assignment 3: Ordered linked list of Items

Total points: 35

Out: February 12, 2025

Due: February 28, 2025

## **IMPORTANT: Read carefully before you start**

1. Start early, ideally on the next day after the assignment is published.
2. Read the whole assignment and make sure you understand the requirements. See the tutor ASAP if you have any questions.
3. Solve the problem in stages. Test every method you implement before implementing the next method.
4. Your assignment will receive ZERO if the submitted code has compilation errors. It is better to receive partial credit for a running program that lacks one or more feature than to receive zero for a program that has compilation errors.
5. Save working versions periodically so you can return to a version that works as needed.
6. Use the same exact method names (case sensitive) and input/output requirements as explained below.
7. Each method’s input parameters must be in the same order as specified.
8. Upload only zip file to the designated D2L folder. Your assignment will receive ZERO if you upload separate files.
9. You may submit more than once. Only the most recent submission will be considered for grading.

# Introduction

The goal of this assignment is to practice:

* using a linked list to implement a collection class.
* using your collection class to implement an application that needs to maintain two separate sets of data.

First, you will implement an ordered collection of Items, called **ChildrenOrderedList**. The collection will store the items in a linked list. The input file of **Items** should not be sorted. You create the sorted list by inserting each item in its proper spot in the list. For example, the first item in your order (e.g., first task in your to-do list) will be stored in is the first node of the linked list and so on. To support the ordering, your list will have methods for inserting/removing/reading items from specific positions in the linked list. You may not use the LinkedList class from the Java library; instead, you must implement your own linked list as explained below.

The order of items in the collection will be based on one or both of the instance variables of the **Children**. It’s your choice how to decide on the order. Your documentation of the co**mpareTo()** method that you use in ordering the **Item**s should clearly explain how you choose to order them.

After you implement the ordered list, you will use this list to implement an application, called **ChildrenTypesLists**, that maintains two ordered lists, one for each subitem type. For example, if your **Item** were a **Pet**, then your **ChildrenTypesLists** would maintain one list for **Dog** and another list for **Cat**.

For this assignment, every Item in the test file must be one of the subitem types. So if your **Item** were a **Pet**, each in the list would have to be either a dog or a cat.

# Requirements

To complete this project, you need to implement the following classes. The details of each class are explained below. The classes in this assignment should be named by replacing **Children** with the actual name of your item. For example, if you choose to implement a collection of pets, then your classes should be named **Pet, Dog, Cat, PetNode, PetOrderedList, PetOrderedListDriver, PetTypesLists**, and **PetTypesListsDriver.**

* **Children**, **GrandChildren**, and **Nephew**: these must be the same item and subitem classes that you used in Assignment 2. If you did not complete Assignment 2, go back and implement only Part 2 of Assignment 2.
* **ChildrenNode**: a linked list node where the data part is of type Item.
* **ChildrenOrderedList**: a linked list of ChildrenNode.
* **ChildrenTypesLists**: a collection class that has two lists of type **ChildrenOrderedList**, one to store items of type **GrandChildren** and the other to store items of type **Nephew**.
* **ChildrenOrderedListDriver**: a Java class that includes only a main method to test the functionality of your **ChildrenOrderedList** class.
* **ChildrenTypesListsDriver**: a Java class that includes only a main method to test the functionality of your **ChildrenTypesLists** class.

## Note: Design note on the **Children** class

Since all Items must be of one or other subitem type (e.g., each **Pet** must be a **Dog** or a **Cat**), theoretically the **Children** class should be declared as an **abstract** class, so that no one can instantiate an **Item** that is not a subitem. However, making the Item class **abstract** would break your existing code for Assignment 2, so you will have to check that each Item is a proper subitem in your code.[[1]](#footnote-1)

## Part 1: Implement **class ChildrenNode**

The **ChildrenNode** class represents a node in the linked list. The class should include the following:

***Instance variables;***

* **private Item data**;

the data item stored at this Node. It is fine to use a term that relates to the Item here instead of the word “data”. For instance, if your Item is a Pet, you might want to say **Pet myPet;**

* **private ItemNode link**

a reference to the next node in the linked list

***Constructor:***

* **public ChildrenNode( Item i, ChildrenNode n** **)**

Initializes the two instance variables.

***Methods:***

* public getters and setters for the instance variables: **getData**, **getLink**, **setData**, **setLink**.

## Part 2: Implement **class ChildrenOrderedList**

The **ChildrenOrderedList** class is a collection class that uses a linked list to store a collection of items and the class supports operations to add/remove/read items from a specific position in the list. The class should include the following:

*Strong suggestion: Create the driver class* ***ChildrenOrderedListDriver*** *(see “Part 3”) at the same time you create this class, and use it to test each method in this class before you go on to the next method.*

***Instance variables:***

* **private ChildrenNode head**

A reference variable to the head of the linked list.

* **private ChildrenNode tail**

A reference variable to the tail of the linked list.

***Constructor:***

* **public ChildrenOrderedList()**

A no-argument constructor to construct an empty list.

***Methods:***

* + **public void add(Item element)**

Adds an Item to the end of the list.

* + **public int size()**

Returns the number of nodes in the list.

* + **public void display()**

Displays the contents of the list, one item per line. You need to display the values of the item’s attributes separated by tab. Do not include any additional text. Note that this method displays the list on the screen and does NOT return a String representation of the list.

* + **public boolean add(int index, Item element)**

Adds an element to the list at the given index where the head node is at index 1. If index is greater than the list size, then the element is added as the last element in the linked list. The method does not do anything if index is less than 1. The method returns true if the item is added and false otherwise.

* + **public boolean remove(Item target)**

Removes one occurrence of target from the list if any. The method returns true if an item is removed and false otherwise. All other items in the list should remain in the same order. Use equals() method to search for target.

* + **public boolean remove(int index)**

Removes the item located at position index in the list where the head node is at index 1. The method returns true if an item is removed and false if no element is removed because index is negative or beyond the list length. This method should not change the order of the other elements in the list.

* + **public int indexOf(Item target)**

Returns the index of the first occurrence of target if any and returns -1 if target is not in the list. Use equals() method to search for target.

* + **public Item get(int index)**

Returns the item at position index in the list and returns null if index is less than 1 or greater than the size of the list.

## Part 3: Implement **class ChildrenOrderedListDriver**

Create a driver class, **ChildrenOrderedListDriver**, and use it to test each method before you implement to the following method. A part of your grade for each method is allocated to the method test from the driver class. This class only needs a main method to test the **ChildrenOrderedListDriver** class, you don’t need to create an object of this class.

## Part 4: Implement **class ChildrenTypesLists**

You will use your **ChildrenOrderedList** collection class in an application that needs to maintain two subsets of items, one subset is of type **GrandChildren** and the other subset is of type **Nephew**.

*Reminder: test all methods in* ***ChildrenOrderedList*** *before you start implementing the method in* ***ChildrenTypesLists****.*

***Instance Variables:***

* **private ChildrenOrderedList GrandChildrenList**

A list that is used to store items of type **GrandChildren**.

* **private ChildrenOrderedList NephewList**

A list that is used to store items of type **Nephew**.

***Constructor:***

* **public ChildrenTypesLists()**

Construct the empty ordered lists for **GrandChildren** and **Nephew** (i.e., initialize the instance variables). You will need to call the **ChildrenOrderedList** constructor to construct each list.

***Methods:***

Implement the following methods. Use the exact method headers as specified. Replace **Item** with your specific item name.

* + **public void add(Item element)**

Adds **element** to the correct list. Find the correct list by using the **instanceof** operator. If the Item is not of subtype **GrandChildren** or **Nephew**, throw a new **ClassNotFoundException** and catch it. When you catch it, terminate the program with the following error message to the console: “Element not instance of either subclass.”

* **public int sizeOf( ChildrenOrderedList iol )**

Returns the number of nodes in the given list.

* **public void display( ChildrenOrderedList iol )**

Displays the contents of the specified list. Note that this method displays the list on the screen and does not return a **String** representation of the list.

* **public boolean add( int index, Item element )**

Adds **element** to the correct list at the given **index** where the head node is at index 1. If **index** is greater than the list size, then the element is added as the last element in the linked list. The method does not do anything if **index** is less than 1, or if element is not of subtype **GrandChildren** or **Nephew**. The method returns **true** if the item is added and **false** otherwise.[[2]](#footnote-2)

* **public boolean remove( Item target )**

Removes one occurrence of **target** if there is at least one instance of **target** in the correct list. The method returns **true** if an **Item** is removed and **false** otherwise. All other **Items** in the list should remain in the same order. Use the **equals()** method to search for **target**.

* **public boolean remove( ChildrenOrderedList iol, int index )**

Removes the **Item** located at position **index** in the **ChildrenOrderedList iol** where the head node is at index 1. The method returns **true** if an element is removed and **false** if no element is removed because index is negative or beyond the list length. This method should not change the order of the other elements in the list.

* **public int indexOf( Item target )**
* Returns the index of the first occurrence of target in the appropriate ordered list. Returns -1 if target is not in the list. Use the equals() method to search for target.
* **public ChildrenOrderedList getList( char type )**

Returns the ChildrenOrderedList of the given type (‘a’ for GrandChildrenOrderedList or ‘b’ for NephewOrderedList). The character should be case-insensitive. Return null if type is invalid.

* **public Item get( ChildrenOrderedList iol, int index )**

Returns the item at position **index** in the specified list. Returns **null** if **index** is less than 1 or greater than the size of the list.

## Part 6: Implement **ChildrenTypesListsDriver**

1. At this point, you should have a driver class that you used to test the methods in the **ChildrenOrderedList**. For this part of the assignment, you need to create another driver class to test methods in **ChildrenTypesLists**.
2. Similar to the previous assignments, your driver should read the data from a text file where each line in the input file represents one item and the line includes the following values separated by tab: (1) a number in {1,2} to specify whether the line represents an **GrandChildren** or **Nephew** object respectively, (2) string value, (3) integer value, and (4) an integer value for **GrandChildren** elements or boolean value for **Nephew** elements. Assume the input file is formatted correctly and you do not have to check for lines that are not in the given format. This is an example input file:

**1 aaa 10 20**

**2 bbb 20 true**

**2 aaa 10 false**

**1 ddd 40 50**

1. Create an input file of at least 20 lines with at 10 lines for each item type (**GrandChildren**/**Nephew**).
2. Write code to ask the user for the path of the data file, read the file name, open the file, read the file line by line, and upload the data to an **ChildrenTypesLists**.
3. Display the two lists. Each list should have at least 10 items.
4. Include at least two test cases for each of the following methods: **indexOf**, **get** (one get from each list). Note that each method from **ChildrenTypesLists** calls the corresponding method from the **ChildrenOrderedList**.
5. The following methods change the list. Include at least two test cases for each method and display after the method call to ensure the method works as expected**: add(index,item**), **remove**(Item), **remove(list, index)**.

## What to submit?

Upload to D2Lonly one zip file that includes the following files:

1. The **.txt** file you used to test your code.
2. **Item.java**
3. **GrandChildren.java**
4. **Nephew.java**
5. **ChildrenOrderedList.java**
6. Driver class for Children**OrderedList**
7. **ChildrenTypesLists.java**
8. **ChildrenOrderedListDriver.java**

## Grading Rubric

The below table shows the order in which your code will be graded, and the number of points assigned to each item. Note that the functionality in each row will be graded only if all previous functionalities are correct. For example, **ItemNode** (row 2) won’t be graded if the classes in row 1 are missing or incorrect. Similarly, display (row 4) won’t be graded if add (row 3) is missing, has compilation errors, or not working as expected.

|  |  |  |
| --- | --- | --- |
| Functionality | | Points |
| 1 | **Item, GrandChildren, Nephew** | **2** |
| 2 | **ItemNode** | **2** |
| **ChildrenOrderedList** | | **16** |
| 3 | **add(Item)** | 2 |
| 4 | **display()** | 2 |
| 5 | **add(int,Item)** | 3 |
| 6 | **remove(Item)** | 2 |
| 7 | **remove(int)** | 3 |
| 8 | **indexOf(Item)** | 2 |
| 9 | **get(int)** | 2 |
| **ChildrenTypesLists** | | **9** |
| 10 | **add(Item)** | 2 |
| 11 | **display** | 1 |
| 12 | **add(int,Item)** | 2 |
| 13 | **remove(Item)** | 1 |
| 14 | **remove(int)** | 1 |
| 15 | **indexOf(Item)** | 1 |
| 16 | **get(int)** | 1 |
| **ChildrenOrderedListDriver** | | **6** |
| 17 | Uploading data to **ChildrenTypesLists** | 3 |
| 18 | There are at least two test cases for each method and the printing of the list is added to check the methods work as expected. | 3 |

1. This is an example of a “design tradeoff”. In order to not break existing code, we don’t do the implementation in the simplest way possible. Another, possibly better way to do this in real life would be to go back to assignment 2, make the Item class abstract, and include a third subclass “Other” for items that are not of type GrandChildren or Nephew. [↑](#footnote-ref-1)
2. Looking ahead, note that the two add methods show two different ways of handling invalid data: throwing an exception or returning false. When we study Queues you will see that the Java API has two methods for adding/removing objects from a queue, one of which throws exceptions and the other of which returns a boolean value. [↑](#footnote-ref-2)