CSCI 2720 – Fall 2025

Programming Assignment 3: Binary Search Trees

Due by 11:59 PM on November 1st, 2025 on Odin

You can complete this assignment individually or as a group of two students.

1 Project Setup

Your source code should be kept under a package `cs2720.p3`. You are free to setup your project using Maven (recommended) or by creating the directory structure manually. Regardless, you must briefly document how to compile and run your project in your README file.

2 Project Description

In this assignment, you will create a Binary Search Tree to store and retrieve objects. Like Programming Assignment 2, you will use generics to make your programs support three different data types (`int`, `double`, and `String`). The data type will be specified by the user before any operations are run, as described later in the document. The purpose of this assignment is for you to become familiar with basic tree operations and understand the efficiency of trees compared to previously studied data structures. Binary Tree nodes have only two children: left and right. Nodes are compared based on their Key instance variable. All elements in the left subtree of a given node have key values less than the given node and all elements in the right subtree have key values greater than the given node. A Binary tree must maintain this property at all times. In this assignment, the binary tree should not accept duplicate values.

You may choose to implement the functions in the Binary Tree class iteratively or recursively. As with the previous assignments, you may create additional functions to implement the features asked in the document. Once all functions have been implemented for the Binary Tree class, you will create a Main application (driver file) that initializes a tree based on the file input and allows the user to interactively modify the tree based on the given commands. Finally, **make sure to properly document all the code with comments, and make sure that your output exactly matches the example output**.

2.1 Java Generics

Like in Programming Assignment 2, you should make your BST and NodeType classes generic. For example, to make NodeType and BinarySearchTree class generic, use the following code snippet:

```
public class NodeType<T extends Comparable<T>> {
    private T info;
    private NodeType<T> left;
    private NodeType<T> right;
} // NodeType
```

```
public class BinarySearchTree<T extends Comparable<T>> {
    private NodeType<T> root;

    public void delete(T key) {
        // ...
    }

    // other methods will go here
} // BinarySearchTree
```

During the program's execution, this "T" will get replaced by the datatype selected by the user for the linked list. You need the `Comparable` class to compare generic objects.

Your program should support storing data of type `int`, `double`, and `String` depending on the input taken from the user at the very beginning of the program. The user should be able to enter 'i' for `int`, 'd' for `double`, or 's' for `String`. Please see the following sample executions:

```
$ java -cp bin cs2720.p3.BinarySearchTreeDriver string-input.txt
Enter list type (i - int, d - double, s - String): s
```

Using Maven:

```
$ mvn exec:java -Dexec.mainClass="cs2720.p3.BinarySearchTreeDriver" \
        -Dexec.args="string-input.txt"
Enter list type (i - int, d - double, s - String): s
```

In the above example, the user has provided 's' as the input. In this case, you are going to create a tree that is going to store `String` items. You will open the input file name provided from the command prompt to read `String` data and store each `String` input in the tree like you did in Assignments 1 and 2. The insert, delete, print, and other commands should likewise support `String` inputs. If the user provides 'i' or 'd', your program should be able to work with `int` or `double` types respectively.

The user is responsible for selecting an appropriate type for a given program run depending on the provided input file and the values that are expected to be stored in the list. You will be provided sample text files for each appropriate data type: one for `int`, one for `double`, and one for `String`.

Similar to Programming Assignments 1 and 2, make sure your code performs error checking for file I/O operations. I.e., it should throw an error message if the file is not present or unable to be opened correctly. You should not hardcode the input file names in the main method.

3 Functional Requirements

A functional requirement is *added* to your point total if satisfied.

Project Files:

- `BinarySearchTreeDriver.java`: Create a main application that matches the sample output.
- `NodeType.java`: Same as Programming Assignment 2, except nodes have left and right children instead of next and back.

- `BinarySearchTree.java`
  - Private instance variables:
    - `private NodeType<T> root;`
  - Public constructors:
    - `public BinarySearchTree()`
      - Pre-Condition: None
      - Post-Condition: Tree is initialized
  - Public member functions:
    - `public void insert(T key)`
      - Pre-Condition: Tree and parameter key are initialized
      - Post-Condition: A node with value key is inserted into the tree. Throw an appropriate runtime exception when trying to insert a duplicate key.
    - `public void delete(T key)`
      - Pre-Condition: Tree and parameter key are initialized
      - Post-Condition: Remove a node with a key value equal to the parameter key's value; otherwise, throw an appropriate runtime exception and leave the tree unchanged (if the key is not present). In situations where the node to be deleted has two children, replace the deleted node with its immediate predecessor or successor.
    - `public boolean retrieve(T item)`
      - Pre-Condition: Tree and item are initialized
      - Post-Condition: item should refer to a key of a Node n in the tree where the value of n.`info` is equal to the value of *item*. If n exists in the tree, return true. Otherwise, throw an appropriate runtime exception.
    - `public void inOrder()`
      - Pre-Condition: Tree is initialized
      - Post-Condition: The tree is printed in order
    - `public boolean isProper()`
      - Pre-Condition: Tree is initialized
      - Post-Condition: Returns whether every node has exactly 0 or 2 children
    - `public boolean isComplete()`
      - Pre-Condition: Tree is initialized
      - Post-Condition: Returns whether every level in the tree is full, or all but the last level is full and the leaf nodes are as far left as possible

1. **The following functions can be implemented however you like, just make sure their input and output formatting matches the sample output below. Implement these functions using method signatures of your choice.**
2. **For the above functions like inOrder, the function prototype does not include a parameter, so you can implement them by using an auxiliary function, using a "getRoot" function, etc.**

`getSingleParent` function – This function should print the nodes that have one child.

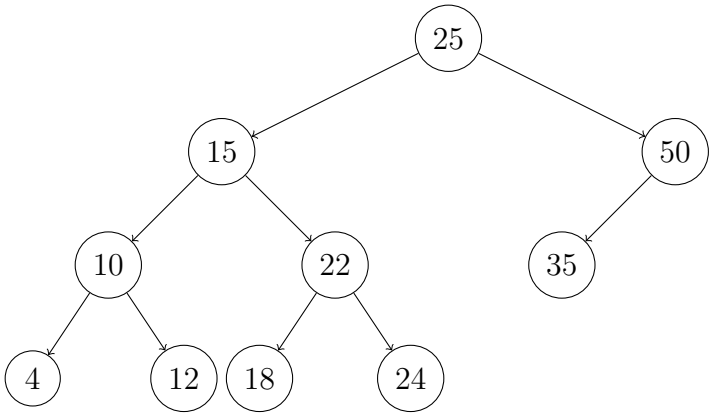
`getNumLeafNodes` function – This function should count the number of leaf nodes in the BST (Nodes with no children) and then output the count.

`getCousins` function – This function should take in a node as input and prints the cousins of the given node.

- Example:
  - In the figure below, 12 has two cousins 18 and 24. Similarly, 35 has two cousins 10 and 22, while 15 and 50 do not have any cousins.

In the README file give the pseudocode (steps) for the above 3 operations. Using this pseudocode to explain the complexity (Big-O) of your 3 operations. To compute the complexity of the above functions, write a recurrence relation and then solve that recurrence relation using the Master method (as discussed in the lectures).

Example:



Sample Outputs

The output of your program should match the sample output below. You don't need to match the exact whitespaces, newline etc. but your output should look like the sample output below. For error checking, use the same error checking that you had used in Programming Assignment 2 (giving a wrong choice for command, checking if filename provided, unable to open the text file, etc.).

For the following examples, the green "// comments" would not be printed by your program.

Blue text is entered by the user.

Sample Output 1 (int-input.txt)

```
Enter list type (i - int, d - double, s - string): i
Commands:
(i) - Insert Item
(d) - Delete Item
(p) - Print Tree
(r) - Retrieve Item
(l) - Count Leaf Nodes
(s) - Find Single Parents
(c) - Find Cousins
(o) - Is Proper
(m) - Is Complete
(q) - Quit program


//1. Print Tree(p) will do the in-order traversal and print the nodes
Enter a command: p
In-order: 4 10 12 15 18 22 24 25 35 50


//2. Insert(i)
//2a. Insert an item that is not present in the tree
Enter a command: i
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number to insert: 20
In-order: 4 10 12 15 18 20 22 24 25 35 50


//2b. Insert an item that is already present in the tree
Enter a command: i
In-order: 4 10 12 15 18 20 22 24 25 35 50
Enter a number to insert: 20
The item already exists in the tree.
In-order: 4 10 12 15 18 20 22 24 25 35 50
```

```
//3. Search/Retrieve(r)
//3a. Search an item that is already present in the tree
Enter a command: r
In-order: 4 10 12 15 18 20 22 24 25 35 50
Enter a number to search: 20
Item is present in the tree


//3b. Search an item that is not present in the tree
Enter a command: r
In-order: 4 10 12 15 18 20 22 24 25 35 50
Enter a number to search: 67
Item is not present in the tree


//4. delete
//4a. Delete a leaf node
Enter a command: d
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number to delete: 4
In-order: 10 12 15 18 22 24 25 35 50
Enter a command: d
In-order: 10 12 15 18 22 24 25 35 50
Enter a number to delete: 24
In-order: 10 12 15 18 22 25 35 50


//4b. Delete a node with one child
Enter a command: d
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number to delete: 50
In-order: 4 10 12 15 18 22 24 25 35
Enter a command: d
In-order: 4 10 12 15 18 22 24 25 35
Enter a number to delete: 18
In-order: 4 10 12 15 22 24 25 35
Enter a command: d
In-order: 4 10 12 15 22 24 25 35
Enter a number to delete: 22
In-order: 4 10 12 15 24 25 35
```

```
//4c. Delete a node with two children
Enter a command: d
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number to delete: 22
In-order: 4 10 12 15 18 24 25 35 50
Enter a command: d
In-order: 4 10 12 15 18 24 25 35 50
Enter a number to delete: 15
In-order: 4 10 12 18 24 25 35 50
Enter a command: d
In-order: 4 10 12 18 24 25 35 50
Enter a number to delete: 25
In-order: 4 10 12 18 24 35 50


//4d. Delete a node that is not present in the tree
Enter a command: d
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number to delete: 33
The number is not present in the tree


//5. Count leaf nodes (it is tested by deleting some of the nodes in between)
Enter a command: p //using given int-input file
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a command: l
The number of leaf nodes are 5
Enter a command: d
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number to delete: 35
In-order: 4 10 12 15 18 22 24 25 50
Enter a command: l
The number of leaf nodes are 5
Enter a command: d
In-order: 4 10 12 15 18 22 24 25 50
Enter a number to delete: 50
In-order: 4 10 12 15 18 22 24 25
Enter a command: l
The number of leaf nodes are 4
Enter a command: d
In-order: 4 10 12 15 18 22 24 25
Enter a number to delete: 18
In-order: 4 10 12 15 22 24 25
Enter a command: l
The number of leaf nodes are 3
```

```
//6. Find single parents (it is tested by deleting some of the nodes in between)
Enter a command: p
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a command: s
Single Parents: 50
Enter a command: d
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number to delete: 18
In-order: 4 10 12 15 22 24 25 35 50
Enter a command: s
Single Parents: 22 50
Enter a command: d
In-order: 4 10 12 15 22 24 25 35 50
Enter a number to delete: 4
In-order: 10 12 15 22 24 25 35 50
Enter a command: s
Single Parents: 10 22 50
Enter a command: d
In-order: 10 12 15 22 24 25 35 50
Enter a number to delete: 12
In-order: 10 15 22 24 25 35 50
Enter a command: s
Single Parents: 22 50
Enter a command: d
In-order: 10 15 22 24 25 35 50
Enter a number to delete: 24
In-order: 10 15 22 25 35 50
Enter a command: s
Single Parents: 50
Enter a command: d
In-order: 10 15 22 25 35 50
Enter a number to delete: 35
In-order: 10 15 22 25 50
Enter a command: s
Single Parents:
```

```
//7. Find cousins
Enter a command: p
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a command: c
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number: 4
4 cousins: 18 24
Enter a command: c
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number: 10
10 cousins: 35
Enter a command: c
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number: 35
35 cousins: 10 22
Enter a command: c
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number: 15
15 cousins:
Enter a command: c
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number: 25
25 cousins:
Enter a command: c
In-order: 4 10 12 15 18 22 24 25 35 50
Enter a number: 22
22 cousins: 35


//8. Is Proper
Enter a command: o
In-order: 4 10 12 15 18 22 24 25 35 50
This tree is not proper.

//9. Is Complete
Enter a command: m
In-order: 4 10 12 15 18 22 24 25 35 50
This tree is not complete.
```

Sample Output 2 (double-input.txt)

```
Enter list type (i - int, d - double, s - string): d
Commands:
(i) - Insert Item
(d) - Delete Item
(p) - Print Tree
(r) - Retrieve Item
(l) - Count Leaf Nodes
(s) - Find Single Parents
(c) - Find Cousins
(o) - Is Proper
(m) - Is Complete
(q) - Quit program


//1. Print Tree(p) will do the in-order traversal and print the nodes
Enter a command: p
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0


//2. Insert(i)
//2a. Insert an item that is not present in the tree
Enter a command: i
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number to insert: 12.9
In-order: 1.3 3.2 10.9 12.9 13.8 15.1 20.4 60.3 89.0


//2b. Insert an item that is already present in the tree
Enter a command: i
In-order: 1.3 3.2 10.9 12.9 13.8 15.1 20.4 60.3 89.0
Enter a number to insert: 3.2
The item already exists in the tree.
In-order: 1.3 3.2 10.9 12.9 13.8 15.1 20.4 60.3 89.0


//3. Search/Retrieve(r)
//3a. Search an item that is already present in the tree
Enter a command: r
1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number to search: 13.8
Item is present in the tree
```

```
//3b. Search an item that is not present in the tree
Enter a command: r
1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number to search: 76.5
Item is not present in the tree


//4. delete
//4a. Delete a leaf node
Enter a command: d
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number to delete: 1.3
In-order: 3.2 10.9 13.8 15.1 20.4 60.3 89.0


//4b. Delete a node with one child
Enter a command: d
In-order: 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number to delete: 15.1
In-order: 3.2 10.9 13.8 20.4 60.3 89.0


//4c. Delete a node with two children
Enter a command: d
In-order: 3.2 10.9 13.8 20.4 60.3 89.0
Enter a number to delete: 20.4
In-order: 3.2 10.9 13.8 60.3 89.0


//4d. Delete a node that is not present in the tree
Enter a command: d
In-order: 3.2 10.9 13.8 60.3 89.0
Enter a number to delete: 17.0
The number is not present in the tree
```

```
//5. Count leaf nodes (it is tested by deleting some of the nodes in between)
Enter a command: p //using given double-input file
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a command: l
The number of leaf nodes are 3
Enter a command: d
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number to delete: 89.0
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3
Enter a command: l
The number of leaf nodes are 3
Enter a command: d
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3
Enter a number to delete: 60.3
In-order: 1.3 3.2 10.9 13.8 15.1 20.4
Enter a command: l
The number of leaf nodes are 2


//6. Find single parents (it is tested by deleting some of the nodes in between)
Enter a command: p
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a command: s
Single Parents: 3.2 15.1 60.3
Enter a command: d
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number to delete: 1.3
In-order: 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a command: s
Single Parents: 15.1 60.3
Enter a command: d
In-order: 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number to delete: 13.8
In-order: 3.2 10.9 15.1 20.4 60.3 89.0
Enter a command: s
Single Parents: 60.3
Enter a command: d
In-order: 3.2 10.9 15.1 20.4 60.3 89.0
Enter a number to delete: 89.0
In-order: 3.2 10.9 15.1 20.4 60.3
Enter a command: s
Single Parents:
```

```
//7. Find cousins
Enter a command: c
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number: 1.3
1.3 cousins: 13.8
Enter a command: c
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number: 15.1
15.1 cousins: 89.0
Enter a command: c
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number: 89.0
89.0 cousins: 3.2 15.1
Enter a command: c
In-order: 1.3 3.2 10.9 13.8 15.1 20.4 60.3 89.0
Enter a number: 10.9
10.9 cousins:
```

Sample Output 3 (string-input.txt)

```
Enter list type (i - int, d - double, s - string): s
Commands:
(i) - Insert Item
(d) - Delete Item
(p) - Print Tree
(r) - Retrieve Item
(l) - Count Leaf Nodes
(s) - Find Single Parents
(c) - Find Cousins
(o) - Is Proper
(m) - Is Complete
(q) - Quit program


//1. Print Tree(p) will do the in-order traversal and print the nodes
Enter a command: p
In-order: Apple Igloo Jam Movie Party Zoo


//2. Insert(i)
//2a. Insert an item that is not present in the tree
Enter a command: i
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string to insert: Joey
In-order: Apple Igloo Jam Joey Movie Party Zoo


//2b. Insert an item that is already present in the tree
Enter a command: i
In-order: Apple Igloo Jam Joey Movie Party Zoo
Enter a string to insert: Joey
The item already exists in the tree.
In-order: Apple Igloo Jam Joey Movie Party Zoo


//3. Search/Retrieve(r)
//3a. Search an item that is already present in the tree
Enter a command: r
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string to search: Joey
Item is not present in the tree
```

```
//3b. Search an item that is not present in the tree
Enter a command: r
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string to search: Jam
Item is present in the tree


//4. delete
//4a. Delete a leaf node
Enter a command: d
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string to delete: Apple
In-order: Igloo Jam Movie Party Zoo


//4b. Delete a node with one child
Enter a command: d
In-order: Igloo Jam Movie Party Zoo
Enter a string to delete: Igloo
In-order: Jam Movie Party Zoo


//4c. Delete a node with two children
Enter a command: d
In-order: Jam Movie Party Zoo
Enter a string to delete: Movie
In-order: Jam Party Zoo


//4d. Delete a node that is not present in the tree
Enter a command: d
In-order: Jam Party Zoo
Enter a string to delete: Joey
Item is not present in the tree
```

```
//5. Count leaf nodes (it is tested by deleting some of the nodes in between)
Enter a command: p
In-order: Apple Igloo Jam Movie Party Zoo
Enter a command: l
The number of leaf nodes are 3
Enter a command: d
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string to delete: Zoo
In-order: Apple Igloo Jam Movie Party
Enter a command: l
The number of leaf nodes are 3
Enter a command: d
In-order: Apple Igloo Jam Movie Party
Enter a string to delete: Party
In-order: Apple Igloo Jam Movie
Enter a command: l
The number of leaf nodes are 2


//6. Find single parents (it is tested by deleting some of the nodes in between)
Enter a command: p
In-order: Apple Igloo Jam Movie Party Zoo
Enter a command: s
Single Parents: Party
Enter a command: d
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string to delete: Zoo
In-order: Apple Igloo Jam Movie Party
Enter a command: s
Single Parents:
```

```
//7. Find cousins
Enter a command: p
In-order: Apple Igloo Jam Movie Party Zoo
Enter a command: c
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string: Apple
Apple cousins: Zoo
Enter a command: c
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string: Jam
Jam cousins: Zoo
Enter a command: c
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string: Zoo
Zoo cousins: Apple Jam
Enter a command: c
In-order: Apple Igloo Jam Movie Party Zoo
Enter a string: Party
Party cousins:
```

Sample Output 4 (proper-complete.txt)

```
Enter list type (i - int, d - double, s - string): i
Commands:
(i) - Insert Item
(d) - Delete Item
(p) - Print Tree
(r) - Retrieve Item
(l) - Count Leaf Nodes
(s) - Find Single Parents
(c) - Find Cousins
(o) - Is Proper
(m) - Is Complete
(q) - Quit program


// Starting state
Enter a command: p
In-order: 10

// Is Proper 1
Enter a command: o
In-order: 10
This tree is proper.

// Is Complete 1
Enter a command: m
In-order: 10
This tree is complete.


// Insert 5

// Is Proper 2
Enter a command: o
In-order: 5 10
This tree is not proper.

// Is Complete 2
Enter a command: m
In-order: 5 10
This tree is complete.
```

```
// Insert 15, 4, 6, 20

// Is Proper 3
Enter a command: o
In-order: 4 5 6 10 15 20
This tree is not proper.

// Is Complete 3
Enter a command: m
In-order: 4 5 6 10 15 20
This tree is not complete.


// Insert 14, 19, 21

// Is Proper 4
Enter a command: o
In-order: 4 5 6 10 14 15 19 20 21
This tree is proper.

// Is Complete 4
Enter a command: m
In-order: 4 5 6 10 14 15 19 20 21
This tree is not complete.
```

Grading Rubric:

Implementation	Grade
Binary Tree	**95%**
Insert	10%
Delete	15%
In-Order	5%
Retrieve	5%
Get Single Parents	10%
Get Num Leaf Nodes	10%
Get Cousins	20%
Is Proper	5%
Is Complete	10%
Using Generics	5%
Driver file, exception handling	**5%**
Total	**100%**

4 Nonfunctional Requirements

A nonfunctional requirement is *subtracted* from your point total if not satisfied.

4.1 Project Structure

Your implementation should match the provided package structure and visibility of each class and method within reason. Use your best judgment when writing your own classes/methods not explicitly listed in this document. (10–100 points)

4.2 Project Dependencies

Your project must exclusively use your `NodeType` class to build the linked structure. You may not use arrays or any Java collections when inserting nodes into or removing nodes from the tree. (100 points)

4.3 Compiling and Running

Code that fails to compile or code that compiles but fails to run will receive a grade of zero. Your program must accept a single command-line argument as a path to the input file.

4.4 README

Commands to run and compile the program should be documented clearly in the README file.

You must include your full name and university email address in your README file. If you are doing a project in a group of two, list the full names and email addresses of all two group members. If you are in a group, you must also describe the contributions of each group member in the assignment. (5 points)

4.5 Style Requirements

All submitted source files must conform to the CSCI 1302 style guide. Please see the guide on eLC for setting up style checks. (5 points for each style violation, up to 20)

A lack of appropriate inline comments may also constitute improper documentation.

4.6 Unit Tests

You are *encouraged* to write unit tests for your project. In CSCI 1302, you may have done this using custom classes, variables, and methods. Here, we recommend using a framework like JUnit to test your linked list features. Like integrating Checkstyle with your build process, testing frameworks provide another convenient way to speed-up your development process.

A sample test file is included on eLC. Although the package for `SortedLinkListTest` is the same as `SortedLinkList`, note that the test file is under `src/test/java/cs2720/p1`, while the main file is under `src/main/java/cs2720/p1`.

Feel free to check out the JUnit 5 documentation for more details.

You should make sure your JUnit and Surefire dependencies are compatible.

See the Project 1 instructions on eLC for more details.

4.6.1 Note on Checkstyle

Unit tests are meant to be descriptive by design, particularly since they are often only a few lines of code. Because of this, you are not explicitly required to Javadoc methods that are given the `@Test` annotation (aka the actual unit tests that are run). Bear in mind that you are still expected to document a longer test whose purpose is not immediately clear.

4.7 Late Submission Policy

Projects must be submitted before the specified deadline in order to receive full credit. Projects submitted late will be subject to the following penalties:

- If submitted 0–24 hours after the deadline 20% will be deducted from the project score.
- If submitted 24-48 hours after the deadline 40% points will be deducted from the project score.
- If submitted more than 48 hours after the deadline a score of 0 will be given for the project.
- Assignment extensions will not be granted, regardless of the situation. If you need extra time, submit up to two days late and consider the late penalty waiver option discussed in the syllabus.

4.8 Submission Notes

You must include your full name and university email address in your README.txt file. If you are doing a project in a group of two, list the full names and email addresses of all two group members. If you are in a group, you must also describe the contributions of each group member in the assignment.

Submit the following files on Odin:

- NodeType.java

- BinarySearchTree.java
- BinarySearchTreeDriver.java
- README.txt

Odin Login Instructions

Projects should be submitted through the School of Computing's Odin server. You can ssh into Odin using your myID credentials. For example,

```
$ ssh abc12345@odin.cs.uga.edu
```

Group Submissions:

****Note: This assignment can be done individually or in a group of two students. The members of the group should be from the same section. Submit a README file with your names, UGA emails and any other information that you want to give the TA to run your code. If you are doing it in a group of two make sure to include the contribution of each student in the README file.**

If you are working in a group of two, your submission folder must be named as follows:

```
LastName1_LastName2_assignment3
```

When you are submitting the folder you should use the submit command as follows:

```
$ submit LastName1_LastName2_assignment3 <odin-submission-destination>
```

Make sure to replace "`<odin-submission-destination>`" with the correct destination from the table on the bottom of this document.

If you are working in a group, you must **submit the assignment from the Odin account of only one of the members** in the group

Individual Submissions:

If you are working individually, your submission folder must be named as follows:

```
LastName_assignment3
```

Submission command should be used as follows:

```
$ submit LastName_assignment3 <odin-submission-destination>
```

Make sure to replace "`<odin-submission-destination>`" with the correct destination from the table on the bottom of this document.

****For all submissions, the folder name and README file are very important in order to be able to locate your assignment and grade it.**

****Note: Make sure you submit your assignment to the right folder to avoid grading delays.**

When using the submit command, first set your current directory of the terminal to the immediate parent of the assignment folder.

You can check your current working directory by typing:

```
$ pwd
```

For example if your assignment is in the following directory,

```
/documents/csci2720/assignments/LastName_assignment3
```

Before executing the submit command, change to the assignment's immediate parent directory:

```
$ cd /documents/csci2720/assignments/
```

Then execute the submit command:

```
$ submit LastName_assignment3 <odin-submission-destination>
```

Make sure to replace "`<odin-submission-destination>`" with the correct destination from the table on the bottom of this document.

To check if your submission command was valid, there should be a receipt file.

```
$ ls LastName_assignment3
```

This should include something like "`rec54321`."

Selecting the Odin Submission Destination

You must select your Odin submission destination using the following mapping based on the section of the course you are registered to.

Section	Class Time	Odin Submission Destination
CRN 42634	TR 11:10 AM	`csci-2720f`