UNIVERSITY AT ALBANY
State University of New York

COLLEGE OF ENGINEERING AND APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

ICSI213/IECE213 Data Structures

Project 04 by Qi Wang

The following parts are included in this document:

- Part I: General project information
- Part II: Project grading rubric
- Part III: Examples on complete a project from start to finish
- Part IV: A. How to test?
  B. Project description

Proper use of the course materials including the source codes:

All course materials including source codes/diagrams, lecture notes, etc., are for your reference only. Any misuse of the materials is prohibited. For example,

- Copy the source codes/diagrams and modify them into the projects.
    - Students are required to submit the **original work** for the projects. **For each project, every single statement for each source file and every single class diagram for each design (if any) must be created by the students from scratch.**
- Post the source codes, the diagrams, and other materials online.
- Others

Part I: General Project Information

- All work is individual work unless it is notified otherwise.
- Work will be rejected with no credit if
  - The work is not submitted via Duifene.
  - The work is late.
  - The work is partially or entirely written in Chinese.
  - The work is not submitted properly.
    - Blurry, wrong files, not in required format, crashed files, etc.
  - The work is a copy or partial copy of others' work (such as work from another person or the Internet).
- Students must turn in their original work. Any cheating violation will be reported to the college. Students can help others by sharing ideas, but not by allowing others to copy their work.
- Documents to be included/submitted as a zipped folder:
  - UML class diagram(s) – created with Violet UML or StarUML
  - Java source file(s) with Javadoc style inline comments
    - Class section,
    - tree node, super BST class, sub BST class, tree iterator class, tree exception class,
    - ADT class data base interface and its implementation,
    - Driver and its helper class.
  - Supporting files if any (For example, files containing all testing data.)
    - Input file containing at least 20 class sections

Lack of any of the required items or programs with errors will result in a low credit or no credit.

- **How to prepare a zipped folder for work submission?**
  - Copy the above-mentioned files into a folder, rename the folder using the following convention:
    [*Your_first_name*][*your_last_name*]*ProjectNumber*
    - For example*, JohnSmithProject02*
  - Zip the folder. A zipped file will be created.
    - For example*,* a file with name *JohnSmithProject01.zip* will be created*.*
  - Submit the zip file on Duifene.
  - You must submit a project in this format. **Submissions not in the required format may be rejected or will result in a low credit or no credit.**

- **Grades and feedback**: Co-instructors will grade. Feedback and grades for properly submitted work will be posted on Duifene. For questions regarding the feedback or the grade, students should reach out to their co-instructors first. Students have limited time/days from when a grade is posted to dispute the grade. Check email daily for the grade review notifications sent from the co-instructors. **Any grade dispute request after the dispute period will not be considered.**

Part II: Project grading rubric

Project 4: (out of 100 points) **Performance Indicator (PI)**	**LEVELS OF PERFORMANCE INDICATORS**				**Points Earned**
	UNSATISFACTORY	**DEVELOPING**	**SATISFACTORY**	**EXEMPLARY**	
PI1: (10 points) UML design	None/Not correct at all. (0)	Visibility, name, and type/parameter type/return type present only for some members for some classes. Some class relationships are incorrect. (<=5)	Visibility, name, and type/parameter type/return type present for each member for all classes with minor issues, class relationships are correct with minor issues. (<=7)	Visibility, name, and type/parameter type/return type present for each member for all classes without issues, class relationships are correct. (<=10)	
	Instructor's Comments:				
PI2(3 points): Comments (Javadoc format) All tags must be included correctly.	None (0)	Some comments are written properly. (<=1)	Most comments are written properly. (<=2)	All comments are written properly. (<=3)	
	Instructor's Comments:				
PI3(2 points): Variable/method naming	Single-letter names everywhere (0)	Many abbreviations (0)	Full words most of the time (<=1)	Full words, descriptive (<=2)	
	Instructor's Comments:				
PI4(10 points): The class: Class Section: a **unique** class number, a course subject, a catalog number, a class title, and a level	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=5)	Most of the required methods are implemented correctly. Have minor issues. (<=7)	All required methods are implemented correctly. Have no issues. (<=10)	
	Instructor's Comments:				
PI5(10 points): Binary Search Tree super class: Operations on the root only	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=5)	Most of the required methods are implemented correctly. Have minor issues. (<=7)	All required methods are implemented correctly. Have no issues. (<=10)	
	Instructor's Comments:				

PI6(10 points): Binary Search Tree sub class: Additional operations on the root or other nodes	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=5)	Most of the required methods are implemented correctly. Have minor issues. (<=7)	All required methods are implemented correctly. Have no issues. (<=10)	
	Instructor's Comments:				
PI7(10 points): A generic tree node: element, left, and right	No methods are designed correctly. (0)	Some of the methods are designed correctly. Have a lot of issues. (<=5)	Most of the required methods are designed correctly. Have minor issues. (<=7)	All required methods are designed correctly. Have no issues. (<=10)	
	Instructor's Comments:				
PI8(10 points): A generic tree iterator: Implements Iterator interface hasNext, next and traversal operations	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=5)	Most of the required methods are implemented correctly. Have minor issues. (<=7)	All required methods are implemented correctly. Have no issues. (<=10)	
	Instructor's Comments:				
PI9(5 points): Tree Exception class: Derived from RuntimeException, a constructor	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=2)	Most of the required methods are implemented correctly. Have minor issues. (<=4)	All required methods are implemented correctly. Have no issues. (<=5)	
	Instructor's Comments:				
PI10(10 points): ADT Class Database Interface	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=5)	Most of the required methods are implemented correctly. Have minor issues. (<=7)	All required methods are implemented correctly. Have no issues. (<=10)	
	Instructor's Comments:				
PI11(10 points): ADT Class Database Implementation	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues.	Most of the required methods are implemented	All required methods are implemented	

		(<=5)	correctly. Have minor issues. (<=7)	correctly. Have no issues. (<=10)	
	Instructor's Comments:				
PI12(10 points): The driver program and its testing/helper class	No methods are tested correctly. (0)	Some of the methods are tested correctly. Have a lot of issues. (<=5)	Most of the required methods are tested correctly. Have minor issues. (<=7)	All required methods are tested correctly. Have no issues. (<=10)	
	Instructor's Comments:				
				Not submitted as a zipped folder	-10
				Total out of 100	
				Total out of 60	

Part III: Examples on complete a project from start to finish

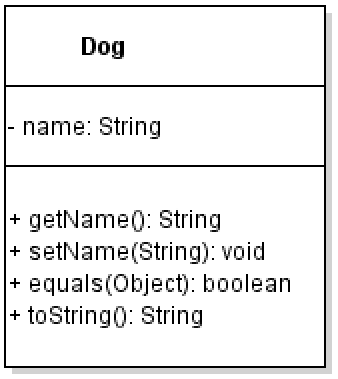
To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

Analysis-design-code-test/debug-documentation.

1) Read project description to understand all specifications (**Analysis**).
2) Create a design (an algorithm for method or a UML class diagram for a class) (**Design**)
3) Create Java programs that are translations of the design. (**Code/Implementation**)
4) Test and debug, and (**test/debug)**
5) Complete all required documentation. (**Documentation**)

The following shows a sample design and the conventions.

- *Constructors* and *constants* should not be included in a class diagram.
- For each *field* (instance variable), include *visibility*, *name*, and *type* in the design.
- For each *method*, include *visibility*, *name*, *parameter type(s)* and *return type* in the design.
    - DON'T include *parameter names*, only *parameter types* are needed.
- Show class relationships such as dependency, inheritance, aggregation, etc. in the design. Don't include the driver program or any other testing classes since they are for testing purpose only.
    - Aggregation: For example, if Class A has an instance variable of type Class B, then, A is an aggregate of B.



The corresponding source codes with inline Javadoc comments are included on next page.

```
import java.util.Random;

/**
 * Representing a dog with a name.
 * @author Qi Wang
 * @version 1.0
 */
public class Dog{
	/**
	 * The name of this dog
	 */
	private String name;

	/**
	 * Constructs a newly created Dog object that represents a dog with an empty name.
	 */
	public Dog(){
		this("");
	}
	/**
	 * Constructs a newly created Dog object wit
	 * @param name The name of this dog
	 */
	public Dog(String name){
		this.name = name;
	}

	/**
	 * Returns the name of this dog.
	 * @return The name of this dog
	 */
	public String getName(){
		return this.name;
	}

	/**
	 * Changes the name of this dog.
	 * @param name The name of this dog
	 */
	public void setName(String name){
		this.name = name;
	}

	/**
	 * Returns a string representation of this dog. The returned string contains the type of
	 * this dog and the name of this dog.
	 * @return A string representation of this dog
	 */
	public String toString(){
		return this.getClass().getSimpleName() + ": " + this.name;
	}

	/**
	 * Indicates if this dog is "equal to" some other object. If the other object is a dog,
	 * this dog is equal to the other dog if they have the same names. If the other object is
	 * not a dog, this dog is not equal to the other object.
	 * @param obj A reference to some other object
	 * @return A boolean value specifying if this dog is equal to some other object
	 */
	public boolean equals(Object obj){
		//The specific object isn't a dog.
		if(!(obj instanceof Dog)){
			return false;
		}
		//The specific object is a dog.
		Dog other = (Dog)obj;
		return this.name.equalsIgnoreCase(other.name);
	}
}
```

Class comments must be written in Javadoc format before the class header. A **description** of the class, author information, and version information are required.

Comments for fields are required.

open {

TAB

TAB

Method comments must be written in Javadoc format before the method header. The first word must be a verb in title case and in the **third** person. Use punctuation marks properly.

open {

TAB

TAB

A description of the method, comments on parameters if any, and comments on the return type if any are required.

A Javadoc comment for a formal parameter consists of three parts:

- parameter tag,
- a name of the formal parameter in the design ,
  (The name must be consistent in the comments and the header.)
- and a phrase explaining what this parameter specifies.

A Javadoc comment for return type consists of two parts:

- return tag,
- and a phrase explaining what this returned value specifies

More inline comments can be included in single line or block comments format in a method.

Part IV:

A. How to test?

There can be many classes in a software design. **A UML class diagram should contain the designs of all classes and the class relationships (For example, is-a, dependency, or aggregation).**

- First, test each class separately in its own driver.
  - Create instances of the class (If a class is abstract, the members of the class will be tested in its subclasses.). For example, the following creates Dog objects.

    Create a default Dog object.
    ```
    Dog firstDog = new Dog();
    ```
    Create a Dog object with a specific name.
    ```
    Dog secondDog = new Dog("Sky");
    ```

  - Use object references to invoke instance methods. If an instance method is a value-returning method, call this method where the returned value can be used. For example, method getName can be called to return firstDog's name. You may print the value stored in firstDogName to verify.

    ```
    String firstDogName;
      …
    firstDogName = firstDog.getName();
    ```

  - If a method is a void method, invoke the method to simply performs the task. Use other method to verify that the method performs the task properly. For example, setName is a void method that changes the name of this dog. After this statement, the secondDog's name is changed to "Blue".

    ```
    secondDog.setName("Blue");
    ```

    Print the return value from getName to verify that the name has been changed correctly.

  - Repeat until all methods are tested.

- Next, test the entire design by creating a driver program and a helper class for the driver program. When completing these two classes, you will test and learn how to use the classes (e.g., Dog class) that you have designed. You may choose different identifiers for the class names. For this project, Driver and Helper are used as the class identifiers.
  - Create a helper class for the driver. The helper class can begin with the following three static methods.

    ```
    public class Helper{
       public static void start(){
               This void method is decomposed into a few tasks:
               Create an empty list.
               Fill the list by calling create with a reference to the list.
               Print the list by calling display with a reference to the list.
              …
       }

       public static returnTypeOrVoid create(a reference to a list) {
               Create objects using data from an input file and store the objects into the list.
                …
       }

       public static returnTypeOrVoid display(a reference to a list) {
               Display the objects of the list.
              …
       }
       …
    }
    ```

- Create a driver program. In *main* of the driver program, call *start* to start the entire testing process.

```
public class Driver{
   public static void main(String[] args){
         Helper.start();
   }
}
```

- There are more methods than what are invoked/tested by the previous three methods. You may add more statements in the methods or write additional methods to test the rest of the methods for all classes.
- The driver and its helper class are for testing purpose only. They should not be included in the design diagram. But you will need to submit their source codes.

B. Project description

For this project, you will create two ADTs:

- A generic *ADT Binary Search Tree*
- An *ADT Class database:* This ADT contains a list of class sections. Therefore, a class representing *a class section* must be created.

B1. A Class Section:

A class section consists of a **unique** class number, a course subject, a catalog number, a class title, and a level. All class sections are sorted by their class numbers. The following shows an example of a class section.

Class Number	**Course Subject**	**Catalog Number**	**Class Title**	**Level**
7122	ICSI	213	Data Structures	Undergraduate

Write a class that represents a *class section*. Test it completely before using it in B3.

B2. A generic ADT Binary Search tree:

A binary search tree is used to store **comparable** objects such comparable class sections. To constrain the type parameter E to be the class types that implement *Comparable* interface, add the interface as the upper bound of E in the class header.

```
public … BinarySearchTree<E extends Comparable<E>>
```

Specifications (Interface):

- **Operations that operate on the root**:
  - Retrieve the root item of this tree.
  - Change the root item of this tree if supported.
  - Check if this tree is empty.
  - Make this tree empty.
- **Additional operations on both the root and other nodes**:
  - Search an element in this tree.
  - Insert an element into this tree.
  - Delete an element from this tree.
  - Construct and return an iterator of *this* tree.

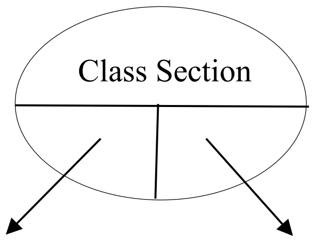
The operations on the root should be included in a super class; the additional operations should be included in its sub class. To support this ADT, the following three classes must be designed and implemented:

- **A generic tree node**: a tree node contains an element, a left reference, and a right reference.
- **A generic tree iterator**: a class implementing *Iterator* interface and provides traversal operations.
- **Tree exceptions**: used for the abnormal situations from some tree operations.

Implement all the operations of this ADT in a class. Constructors should be created in the class. Test it before using this class in the next part, ADT class database.

B3. ADT Class Database:

An ADT class database maintains a list of class sections in a binary search tree. In the binary search tree, the element of a node is a class section. To implement this ADT, you must choose a binary search tree as the data structure. You must use the binary search tree from B2. It is not allowed to use any other BST related classes from the standard library.



Specifications (Interface):

Note: most of the operations are executed in the underlying binary search tree of this class database.

- Insert a class into a class database (You can simply call *insert* via the underlying binary search tree.).
- Delete a class from this class database (You can simply call *delete* via the underlying binary search tree.).
- Search the class in this class database(You can simply call *search* via the underlying binary search tree.).
- Check if this class database is empty.
- Make this class database empty.
- Sort the nodes of this tree: traverse this class database in an inorder traversal, store the class sections in a linked list and return a reference of the list. You can use *java.util.LinkedList* class for this method.
- Return the total number of class sections.

Implement all the operations of this ADT in a class. Constructors should be created in the class.

Now it is time to test the ADT class database. Write a driver and a helper class for the driver. Start with an empty class database, add class sections to the database, and display the class sections. The following gives some ideas for you to start. Add more methods or more statements in the existing methods to make sure that all classes are tested.

```
public static void start(){
    Create an empty class database, list.
    Pass list to create that fills list with class sections*.
        create(list);
    Pass list to display that prints the class sections in list.
        display(list);
            …
}
public static void create(…){…}
```

```
    public static void display(…){…}
```

*To make class section objects needed by *create* method, an input file must be created first. Go to the university class schedule, find at least 20 various classes, and save them in a plain text file. The following shows a couple of classes.

Class Number	Course Subject	Catalog Number	Class Title	Level
7122	ICSI	213	Data Structures	Undergraduate
9797	ICSI	311	Principles of Programming Languages	Undergraduate
9397	AAFS	209	Black American Music	Undergraduate
3980	HHPM	645	Global Health	Graduate