ECE244 Programming Fundamentals

Lab Assignment #6: Student-Marks Database Revisited

1. Objectives

The objectives of this assignment are (1) to provide you with more practice on the use of dynamic memory allocation, pointers, and dynamic data structures; and (2) to provide you with more practice on the use of the `Make` utility and the debugger. This will be done in the context of re-implementing the simple student-marks database of Assignment 5.

2. Problem Statement

In this assignment, you will re-implement Assignment 5's simple database, which was used to store and retrieve student marks in a school term. You will essentially implement two classes: `studentRecord` and `studentDB`. The `studentRecord` class will be used to create objects that hold a single student's information. The `studentDB` class will be used to create a database of `studentRecord` objects. You will implement a tree-based version of the database.

2.1 The `studentRecord` Class

The `studentRecord` class has fields to represent the student number, student last and first names, and 5 marks. It also has the following methods associated with it.

- `studentRecord ()`. This is the default constructor. It creates an empty student record.
- `~studentRecord ()`. This is the destructor. It `delete`s all dynamic components of the student record.
- `void setStudentNumber (unsigned int studentNum)`. This method sets the student number of the student record to `studentNum`.
- `void setFirstName (string firstName)`. This method sets the student first name of the student record to the value of the string `firstName`.
- `void setLastName (string lastName)`. This method sets the student last name of the student record to the value of the string `lastName`.
- `void setMark (int index, unsigned int mark)`. This method sets the mark at index `index` of the student record to `mark`. The marks are indexed 0 to 4. The method does not check that `index` is in this range; the caller must do so. A mark is between 0 and 100, and the caller must also check this.
- `unsigned int getStudentNumber ()`. This method returns the student number of the student record.

- `string getFirstName ()`. This method returns a string containing the first name of the student record.

- `string getLastName ()`. This method returns a string containing the last name of the student record.

- `Unsigned int getMark (int index)`. This method returns the mark at index `index` of the student record. The marks are indexed 0 to 4. The method does not check that `index` is in this range; the caller must do so. A mark is between 0 and 100.

- `void print ()`. This method prints the student record to the standard output, in the following format:

  Student number: `number` ↵
  Student name: `lastname`, `firstname` ↵
  Student marks: `m1`, `m2`, `m3`, `m4`, `m5`

  The student number should be printed as a 9-digit integer. The ↵ character indicates `newline` (i.e., `endl`). Note its absence after the last list of the output.

2.1 The `studentDB` Class

The database has fields to represent the structure of the database and the following methods associated with it:

- `studentDB ()`. This is the default constructor. It creates an empty database.

- `~studentDB ()`. This is the destructor. It deletes all the records in and the structures of the database.

- `bool insert (studentRecord * newRecord)`. This method inserts the record pointed to by `newRecord` into the database. If a record with the same student number already exists in the database, `false` is returned. The database must not be full when this method is called.

- `bool retrieve (unsigned int studentNum, studentRecord * searchRecord)`. This method searches the database for a record with a student number `studentNum`. If the record is found, then its contents are copied into the student record `searchRecord` (which must exist before the method is invoked), and `true` is returned. Otherwise, a `false` is returned and the contents of `searchRecord` are unaffected.

- `bool remove (unsigned int studentNum)`. The method deletes the record with student number `studentNum` from the database. If the record is found and deleted, `true` is returned. Otherwise `false` is returned.

- `void clear ()`. This method clears the database by deleting all the student records in the database, effectively returning the database to its initial empty state.

- `bool isEmpty ()`. This method returns `true` if the database is empty, otherwise, it returns `false`.

- `bool isFull ()`. This method returns `true` if the database is full, otherwise it returns `false`.

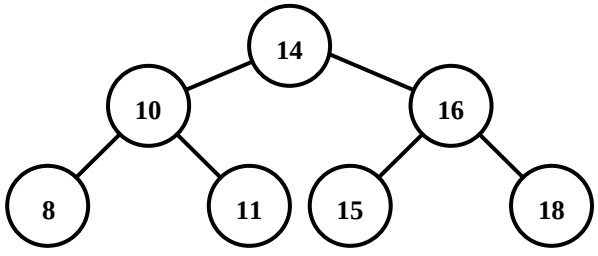
- `void printProbes ()`. This method writes to the standard output the number of ***probes*** performed by the last invocation of the `retrieve` method. A probe is defined as a comparison operation between the search key (i.e., student number) and a key in the database. For example, in the above figure, it takes 2 probes to retrieve the data in the left child of the root node. One comparison it made at the root, another is made at the left child. The number of probes is proportional to the time it takes to search for the search key.

- `void dump ()`. This method dumps out the database to the standard output, one student record at a time (using the `studentRecord`'s `print` function), separated by empty lines, and sorted in ascending order of student numbers.

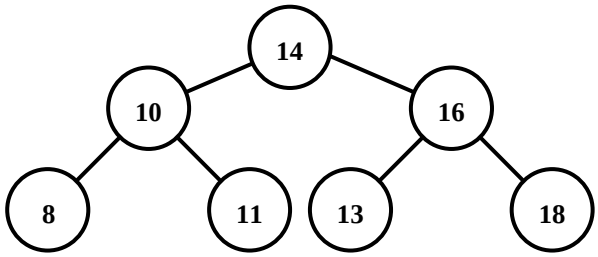
3. Background

A binary search tree (BST) is a binary tree for which: (1) each node has a key that is unique; (2) the key of each node is greater than the keys of all nodes in its left subtree; and (3) the key of each node is smaller than the keys of all nodes in its right subtree. Thus, a BST is an ordered binary tree in which left is smaller than node, which in turn is smaller than right. Since this order property holds for every node in the tree, the left and right subtrees of any node are also binary search trees. That is, the order property holds recursively.

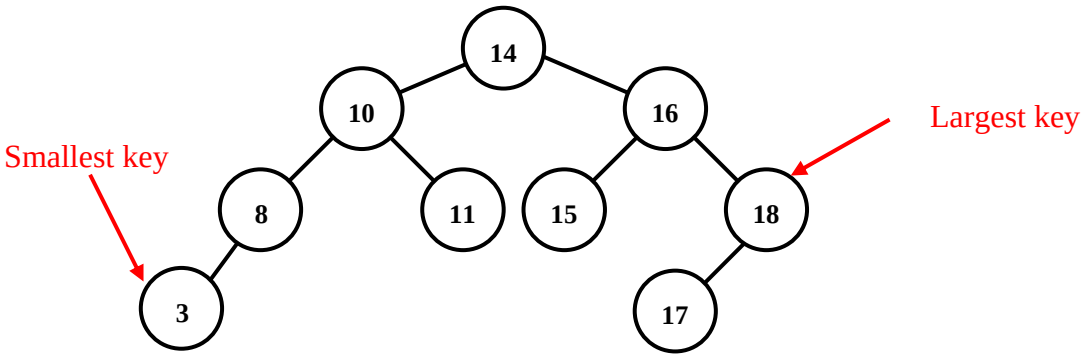
For example, the binary tree below is a BST, in which the keys of the nodes are integers. The key of the root is larger than any of the keys in its left subtree and is smaller than any of the keys in its right subtree. Each of the left and right subtrees is also BST. The left subtree is rooted at a node whose key 10 is larger than all the keys in its own left subtree (8), but smaller than all the keys in its own right subtree (11).



On the other hand, the binary tree below is not a BST (why? Hint: hotel elevators often skip this floor).

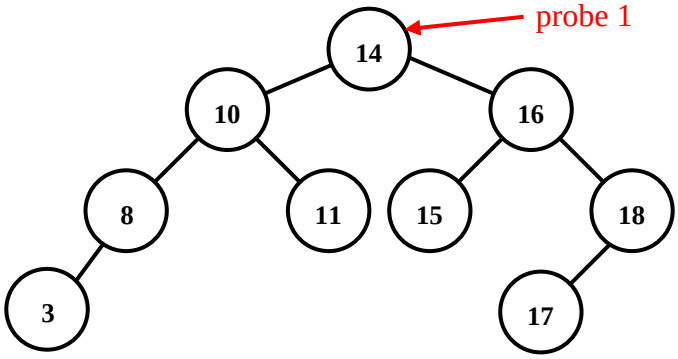


The order property of the BST also allows for sorting the keys of the nodes in a BST. Traversing a BST using the inorder traversal order produces the keys in the tree sorted in ascending order. This is because the inorder traversal always visits the left subtree, followed the node, and then the right subtree (recursively). Indeed, the node in a BST with the smallest key is the left-most one. Similarly, the one with the largest key is the right-most node in the tree, as shown in the example below.

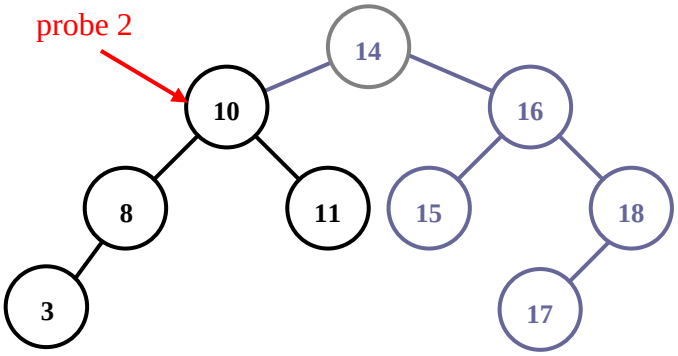


Inorder traversal: 3 8 10 11 14 15 16 17 18

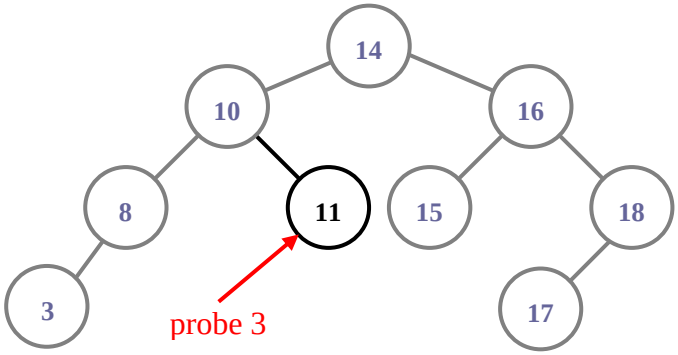
The order property of a BST makes it easy and efficient to search for a node in the tree, with a given search key. Given a BST and a search key ***k***, the goal is to determine if there exists a node in the BST that has a key ***k*** (remember, because the keys are unique, there can be at most one such node). One simply compares the search key ***k*** to the key of the root of the BST. If they are equal, then the root is the node searched for. If ***k*** is less than the key of the root, then if a node exists with a matching key, the node must be in the left subtree. One then recursively repeats the search in the left subtree. Otherwise, ***k*** is greater than the key of the root, and thus, if a node exists with a matching key, it must be in the right subtree. Thus, one recursively repeats the search in the right subtree. Each comparison of the search key with a key of a node in the tree is called a ***probe***. The process of searching for a node with the key 11 in an example BST is illustrated below.



Compare search key 11 to 14. Since 11 < 14, look in the left subtree.



Compare search key 11 to 10. Since 11 > 10, look in the right subtree.

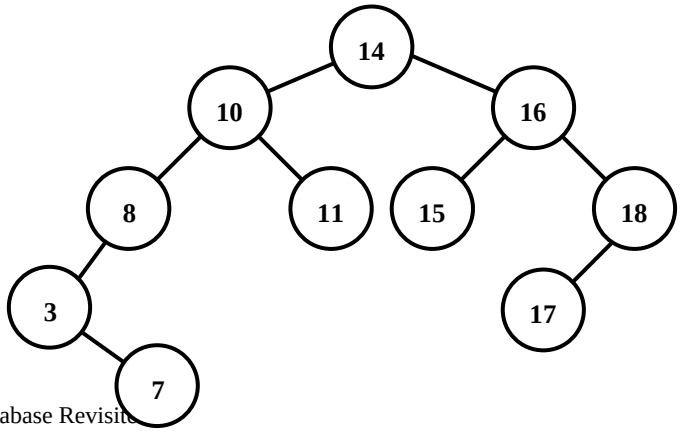


Compare search key 11 to 11. Since 11 = 11, the node with key equal to the search key is found.

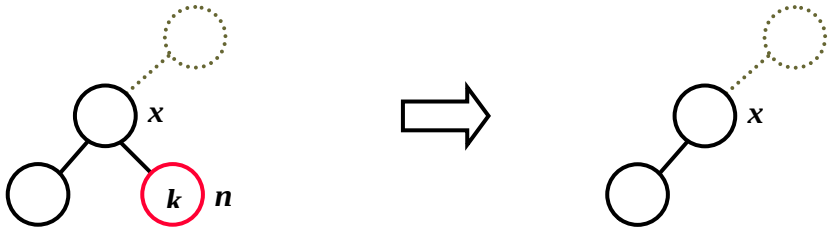
Had the search key been 7, then four probes would have been needed. The first compares 7 to 14, the second compares 7 to 10, the third compares 7 to 8, and the fourth compares 7 to 3. At this point, the search should examine the right subtree of the node with key 3. However, since such subtree does not exist, the search concludes that no node with key 7 exists in the BST.

Inserting new nodes on a BST must be performed in a way that maintains the order property of the BST. Given a BST and a new node ***n*** whose key is ***k***, the goal is to insert the node ***n*** in such a way that the tree remains a BST. It is assumed that no node exists in the tree with a key ***k***. The process is very similar way to searching. The key of the new node (i.e., ***k***) is compared to the key of the root. If ***k*** is less than the key of the root, the new node must be inserted in the left subtree. Otherwise, the new node must be inserted in the right subtree. This is repeated recursively until either no left subtree or no right subtree exists. The new node is inserted there.

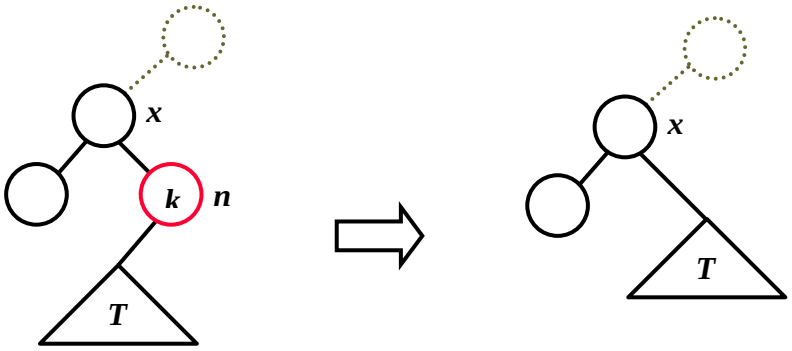
Thus, if a new node with key 7 is to be inserted on the same example BST used to illustrate searching above, the same steps of searching for a node with key 7 would be performed. When the search concludes with failure because of the lack of a right subtree of the node with key 3, the new node is inserted as the right child of this node, as shown below.



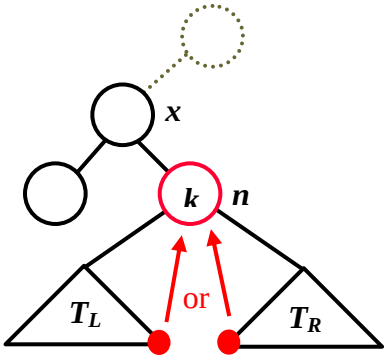
Deleting a node from a BST must also be done in a way that maintains the order property of the BST. Given a BST and a node ***n*** with key ***k*** in the BST, the goal is to delete the node ***n*** in such a way that the tree remains a BST. The actions taken after the node is deleted depend on the location of the node in the tree. If the node ***n*** is a leaf, then the node is just deleted, with no further actions necessary. This is shown in the figure below.



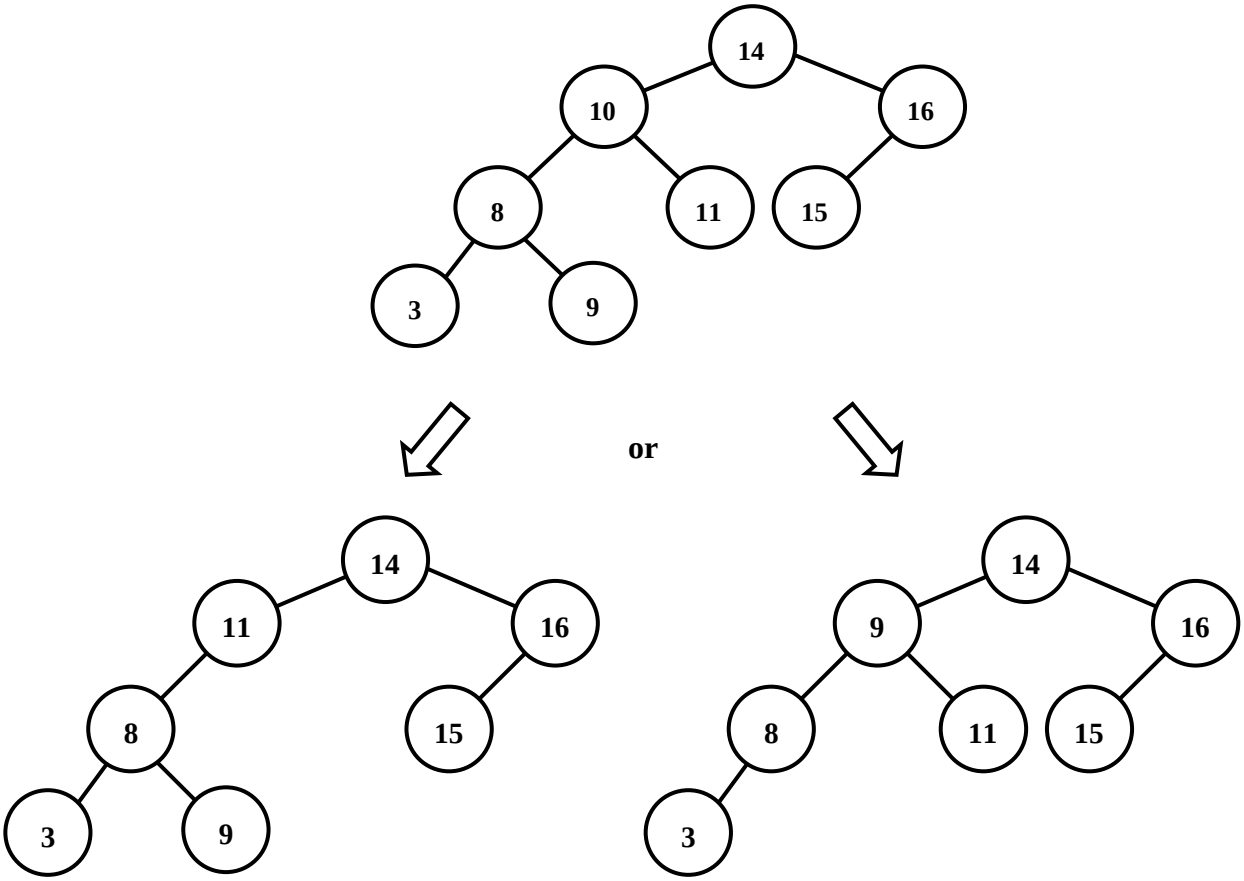
If the node n is not a leaf, but it has only one subtree ***T*** (either left or right), the node ***n*** is removed from the tree, and the its subtree ***T*** is then "connected" to ***n***'s parent (***x***), as shown in the figure below. In this example, ***n*** has only a left subtree ***T***. When ***n*** is deleted, ***T*** must be re-attached to the BST. Since all the nodes in ***T*** have keys that are less than the key of ***n*** (i.e., ***k***), and since ***x***, the parent of ***n***, has a key which is less than ***k***, making ***T*** the left subtree of ***x*** (i.e., making the root of ***T*** the right child of ***x***) maintains the order property of the BST (key(***x***) < key (***i***), where ***i*** is any node in ***T***).



Finally, if the node to be deleted has two subtrees, then when the node is deleted, a node from one of the two subtrees replaces it. The replacement node is selected to maintain the order property of the BST. If the replacement node is selected from the left subtree of the deleted node ***n***, then it must be the node with the largest key in this subtree (i.e., right most as explained above); call it ***l***. This way when ***l*** replaces n, the key of ***l*** is greater than the key of any node its now left subtree, and at the same time smaller than the key of any node in its now right subtree. This is shown the in figure below. Similarly, if the replacement node is selected from the right subtree of the deleted node ***n***, then it must be the node with the smallest key in this subtree (i.e., the left most as explained above); call it ***s***. This way when ***s*** replaces ***n***, the key of ***s*** is smaller than any node in its now right subtree, and at the same time larger than the key of any node in its now left subtree. Thus, irrespective of which subtree to select a replacement node from, the order property of the BST is preserved.



Thus, in the example BST shown below, deleting the node with the key 10 can be done in the two ways shown; by replacing the deleted node with the node with the key 11, or the node with the key 9.



4. Preparation

You must also work on the assignment ***before*** you come to the lab. You should prepare an initial implementation of both classes: `studentRecord` and the tree-based `studentDB` (Parts I and II below) before you arrive to the lab session of the first week of the assignment. You should prepare an initial implementation of the `Driver.cpp` (Part III) before you arrive to the lab session of the second week of the assignment.

You can (and should) re-use the code you developed in Assignment 5 for this assignment.

5. Procedure

Create a sub-directory called `lab6` in your `ece244` directory, using the `mkdir` command. Make it your working directory. ***You may modify any of the .h files that you will download for this assignment only by adding private function members. You may NOT add data members nor public function members***!

5.1 Part I – The Student Record

Use a browser to download the file `studentRec.h`. It contains the definition of the class `studentRecord`, which is intended to hold one student's information. The purpose of the fields and purpose of function members is described in this file. ***You may modify this file only as described above; i.e., only by adding private function members. You may NOT add data members nor public function members***.

Write the implementation of this class in a file called `studentRec.cpp`.

Write a short program that serves as a test-harness for this class, and call it `test-studentRec.cpp`. For example, you may want to write a short program that prompts the user for student numbers, student names, and student marks, then create a `studentRecord` object, and then print it using the `print` method.

Write a `Makefile` that can be used to generate the executable of the test-harness. The `Makefile` should include lines that look like:

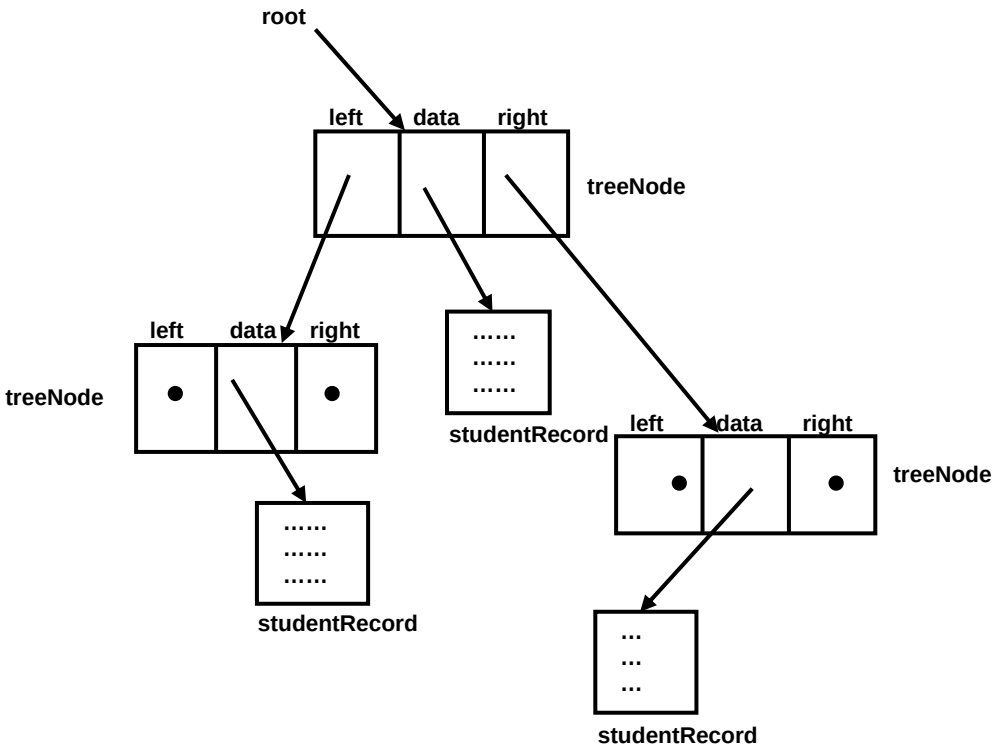
```
test-studentRec:    test-studentRec.o studentRec.o
          g++ test-studentRec.o studentRec.o –o test-studentRec

test-studentRec.o:  test-studentRec.cpp studentRec.h
          g++ -c test-studentRec.cpp
```

You can make the test-harness by typing:

```
make test-studentRec
```

5.2 Part II – The Tree-based Database

The tree-based implementation assumes that the database is made of a BST, with data being pointers to `studentRecord`s, as shown below.



The `left` and `right` pointers fields in each `treeNode` are pointers to the left and right subtrees respectively. The BST is "sorted" by student numbers. That is, the `studentNumber` field of each `studentRecord` is they key of the BST.

Use a browser to download the files `treeNode.h` and `treeDB.h`. They contain the definitions of the class `treeNode` and the class `studentDB`, respectively. ***You may modify these file only as described above***. Write the implementation of these classes in two files, called `treeNode.cpp` and `treeDB.cpp`, respectively.

Write a short program that serves as a test-harness or this class, and call it `test-treeDB.cpp`. Extend your `Makefile` from Part II, to generate the executable of this test-harness.

5.3 Part III – The Driver Program

In this part of the assignment, you will write the user interface to the database in the `main` function. The function should first create an empty database. It should then input from `cin` a sequence of commands. Each command consists of an operation, followed by its parameter. The command and the parameters are separated by white spaces. Your function should process commands until the end-of-file (`eof`) is encountered. The commands and their parameters are:

- **`new`** *`num`*. This command creates a new student record with *`num`* as the student number. The last name and first name fields are initialized to "`LNU`" and "`FNU`", respectively. All marks are initialized to 0's. The new record is then inserted into the database. If there already exists a record with the same number, the error message "`Error: a record with student number num already exists.`" is printed to `cout`. If the database if full, then the error message

"`Error: database is full.`" is printed to `cout`. Otherwise no output is produced.

- **`locate`** *`num`*. This command locates the record with the student number *`num`* in the database, and prints it contents to `cout`. If no such record exists, the error message "`Error: no record with student number` *`num`* `exists.`" is printed to `cout`.

- **`updatename`** *`num lname fname`*. This command locates the record with the student number *`num`* and updates its last name and first name fields with *`lname`* and *`fname`*, respectively. If no such record exists, the error message "`Error: no record with student number` *`num`* `exists.`" is printed to `cout`. If the operation is successful, the contents of the updated record are printed to `cout`.

- **`updatemark`** *`num idx mark`*. This command locates the record with the student number *`num`* and updates its mark at index *`idx`* with *`mark`*. If no such record exists, the error message "`Error: no record with student number` *`num`* `exists.`" is printed to `cout`. If the operation is successful, the contents of the updated record are printed to `cout`. Further, *`idx`* must be in the range 0-4, otherwise the error message "`Error:` *`idx`* `is out of the range 0-4.`" is printed to `cout`. Also, *`mark`* must be in the range 0-100, otherwise the error message "`Error:` *`mark`* `is out of the range 0-100.`" is printed to `cout`.

- **`delete`** *`num`*. This command deletes the record with the student number *`num`* in the database. If no such record exists, the error message "`Error: no record with student number` *`num`* `exists.`" is printed to `cout`.

- **`printall`**. This command prints all the records in the database, sorted in ascending order of student number, separated by spaces.

- **`printprobes`** *`num`*. This command locates the record with the student number *`num`* in the database. If no such record exists, the error message "`Error: no record with student number` *`num`* `exists.`" is printed to `cout`. Otherwise, the number of probes (as defined above) is printed to `cout`.

- **`deleteall`**. This command deletes all the records in the database, retuning it to the empty state.

The following is an example of commands and their outputs. The example assumes an empty database at the beginning. The output is shown in red.

```
% ./treeDriver
new 123456789
locate 987654321
Error: no record with student number 987654321 exists.
locate 123456789
Student number: 123456789
```

```
Student name: LNU, FNU
Student marks: 0, 0, 0, 0, 0
updatename 123456789 Abdelrahman Tarek
Student number: 123456789
Student name: Abdelrahman, Tarek
Student marks: 0, 0, 0, 0, 0
updatemark 123456789 2 500
Error: 500 is out of the range 0-100.
updatemark 123456789 2 99
Student number: 123456789
Student name: Abdelrahman, Tarek
Student marks: 0, 0, 99, 0, 0
updatemark 123456789 5 98
Error: 5 is out of the range 0-4.
updatemark 123456789 4 98
Student number: 123456789
Student name: Abdelrahman, Tarek
Student marks: 0, 0, 99, 0, 98
delete 987654321
Error: no record with student number 987654321 exists.
new 123456789
Error: a record with student number 123456789 already exists.
new 123454321
updatename 123454321 Voss Michael
Student number: 123454321
Student name: Voss, Michael
Student marks: 0, 0, 0, 0, 0
updatemark 123454321 1 57
Student number: 123454321
Student name: Voss, Michael
Student marks: 0, 57, 0, 0, 0
printall
Student number: 123454321
Student name: Voss, Michael
Student marks: 0, 57, 0, 0, 0

Student number: 123456789
Student name: Abdelrahman, Tarek
Student marks: 0, 0, 99, 0, 98

delete 123454321
printall
Student number: 123456789
Student name: Abdelrahman, Tarek
Student marks: 0, 0, 99, 0, 98

delete 123456789
(eof)
```

Extend your `Makefile` from the previous, to generate the executable of this driver. You must test your driver with tree based implementations of the `studentDB` class. You must call the tree based driver `treeDriver.cpp`, with an executable called `treeDriver`.

6. Deliverables

Submit the following components of your implementations: `treeNode.h`, `treeNode.cpp`, `studentRec.h`, `studentRec.cpp`, `treeDB.h`, `treeDB.cpp`, `treeDriver.cpp` and `Makefile` using the `submitece244f` command as follows:

```
submitece244f 6 studentRec.h studentRec.cpp treeNode.h treeNode.cpp
                treeDB.h treeDB.cpp treeDriver.cpp Makefile
```