Programming Project Two: A Simple Database Using 2-3 Trees

Out: Nov. 16, 2012; Due: Nov 30., 2012.

I. Motivation

1. To give you experience in implementing a 2-3 tree ADT.
2. To implement a simple database system using the 2-3 tree ADT.

II. Programming Assignment

You will first implement a templated 2-3-Tree ADT. Then, you will use the 2-3-Tree ADT to build a simple database system.

1. The 2-3-Tree ADT

The 2-3-Tree ADT is a templated class. It is based on the following templated `Node` struct:

```
template <class Key, class Val>
struct Node
{
    Key *lkey;    // left key
    Val *lval;    // left val
    Key *rkey;    // right key
    Val *rval;    // right val
    Node *left;   // left child
    Node *center; // center child
    Node *right;  // right child
};
```

Note that there are two type variables in the template declaration: `Key` and `Val`. They correspond to the actual key type and the value type that will be stored in this node type. For

example, to instantiate a `Node` type with key type as `int` and value type as `string`, you would declare:

```
Node<int, string> node;
```

Instead of storing objects of `Key` type and objects of `Val` type, the `Node` struct keeps pointers to objects of `Key` type and pointers to objects of `Val` type. The `Node` struct is used to represent both 2-nodes and 3-nodes in the tree. The representation specifications are:

(1) For a 3-node, `lkey` points to the smaller key (or the key on the left) and `rkey` points to the larger key (or the key on the right). `lval` and `rval` point to the elements associated with `lkey` and `rkey`, respectively. Pointers `left`, `center`, and `right` point to the left, center, and right subtrees of that node, respectively.

(2) For a 2-node, the only key of the node is pointed by `lkey` and the only element of the node is pointed by `lval`. Both `rkey` and `rval` are not used, so we set them as `NULL`. Pointers `left` and `center` point to the "left" and the "right" subtrees of the 2-node, respectively. Pointer `right` is not used, so we set it as `NULL`.

Our templated 2-3 tree class `Tree` is based the `Node` struct. Its definition is

```
template <class Key, class Val>
class Tree
{
    Node<Key, Val> *root; // The root of the 2-3 tree

public:
    Node<Key, Val> *Root() const { return root; }

    bool isEmpty() const { return !root; }
    // EFFECTS: return true if and only if the 2-3 tree is empty.

    Val *search(Key key) const;
    // EFFECTS: search a record with key as "key" in the 2-3
    // tree. If such a record exists, return the Val*
    // part of the record. Otherwise, return NULL.

    bool insert(Key *key, Val *val);
```

```
    // MODIFIES: this tree
    // EFFECTS: insert (key, val) pair into the 2-3 tree.
    // Return true if insertion succeeds (i.e., no duplicated
    // key exists in the tree). Otherwise, return false.

    Val *remove(Key key);
    // MODIFIES: this tree
    // EFFECTS: remove a record with key as "key" from the
    // 2-3 tree. If such a record exists, return the Val* part
    // of that record. Otherwise, return NULL.

    void inOrderPrint(ostream &os) const;
    // MODIFIES: os
    // EFFECTS: in-order traverse the 2-3 tree and print to the
    // output stream os the Val part of each record visited,
    // with one record per line. To print a record stored in
    // a node at level k, you should first print 4*k spaces and
    // then the Val part of the record. Assume that the root is
    // at level 0.

    Tree();                                 // default constructor
    Tree(const Tree &t);                    // copy constructor
    Tree &operator=(const Tree &t);         // assignment operator
    ~Tree();                                // destructor
};
```

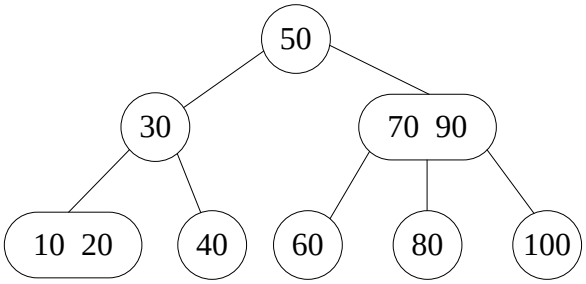
Similar as the `Node` struct, the template declaration of the `Tree` class also depends on two type variables: `Key` and `Val`. The `Tree` class has a single data member `root` storing the root of the tree. It has two simple public methods `Root()`, which returns `root`, and `isEmpty()`, which tells if the tree is empty. You are going to implement the remaining four public methods shown in the class definition: `search()`, `insert()`, `remove()`, and `inOrderPrint()`. The specification of each function is shown above in the comments. You can assume that:

(1) For the `Key` class, it has overloaded the comparison operators <, <=, >, >=, ==, and !=.

(2) For the `Val` class, it has overloaded the stream insertion operator <<.

For the `insert` and the `remove` function, you may find that for some cases you have multiple options on how to modify the tree to keep the 2-3 tree properties. However, for grading purpose, **you should implement the insertion/removal procedure in the way described in our slides.** For example, during removal, if the root of the center subtree of a 3-node becomes empty and both its left and right siblings are 2-node, you could merge the empty node with either its left sibling or its right sibling. However, the rules in our slides state that the empty node should be merged with its left sibling. Only by sticking to the rules specified in our slides could the output of a correct program be the same as ours.

The function `inOrderPrint` in-order traverses the 2-3 tree and prints the `Val` part of each record visited, with one record per line. We require the following format in printing: to print a record stored in a node at level k, you should first print 4*k spaces and then the `Val` part of the record. We assume that the root is at level 0. For example, consider a set of (`Key`, `Val`) pairs {(10, aa), (20, bb), (30, cc), (40, dd), (50, ee), (60, ff), (70, gg), (80, hh), (90, ii), (100, jj)}. Suppose that they are stored in the following 2-3 tree (only keys are shown in the figure).



Calling `inOrderPrint` will output as follows:

```
        aa
        bb
    cc
        dd
ee
        ff
    gg
        hh
    ii
        jj
```

Besides the four public methods, you also need to implement the common four maintenance methods, i.e., the default constructor, the copy constructor, the overloaded assignment operator, and the destructor. However, there is one difference compared with our first programming assignment which requires deep copy. The tree nodes only keep the pointers to the keys and the values; they do not own the key objects and the value objects. The key objects and the value objects are actually maintained by the other part of the program. This means that the destructor should only free the nodes in the tree; it has no permission to free the key objects and the value objects each node keeps! Similarly, the copy functions of the `Tree` class should only make a copy of each node and then copy the pointer values to the keys and the values; it should not further make deep copies of the key objects and the value objects.

In summary, you are going to write eight member functions of the `Tree` class. **You should put your implementations in a file called `two-three-tree.C`** (Note: CAPITAL C!) **exactly**. The definitions of the `Node` struct and the `Tree` class are put in a file named `two-three-tree.h`, which is available in the Assignments/Programming-Project-Two folder on Sakai. You should not modify the file `two-three-tree.h`.

In implementing the above eight functions, you probably need to write some helper functions. A good practice is to declare these helper functions as the private members of the `Tree` class. However, we also observe that these helper functions usually take an argument of `Node` type and the `Tree` methods call these functions by passing `root` to this argument. Therefore, you can define these helper functions as plain functions (i.e., not the private member functions of the `Tree` class). You are required to put these helper functions in the `two-three-tree.C`. This gives you full freedom in defining the helper functions. To prevent compiling error when we link your source code with our test driver code, you should define these helper functions as `static` functions. For example, suppose you need a helper function for search. You can define it as

```
template <class Key, class Val>
static Val *search_help(Node<Key, Val> *node, Key key)
{
  …
}
```

The `search` function of the `Tree` class could then call the helper function as

```
search_help(root, key);
```

We will test your 2-3 tree implementation separately from the other component of this project, so it must work independently of the application described below.

To compile a program that uses the `Tree` class, you need to include both `two-three-tree.h` and `two-three-tree.C`. **To prevent multiple definitions of the class methods, you should use `#ifndef`, `#define`, `#endif` preprocessor directives in the `two-three-tree.C`. You do not need to compile the source file `two-three-tree.C`**, because you already include it in other codes where the 2-3-tree ADT is used.

Reference: The first reference is our course slides. As we have said above, you should stick to the rules specified in our slides when modifying the 2-3 trees. The second reference is Chapter 10.4 of the book by Clifford Shaffer: "Data Structures and Algorithm Analysis," which is online available at

http://people.cs.vt.edu/~shaffer/Book/C++3e20120605.pdf

2. A Simple Database

You are going to implement a simple database system with multiple key indices. Specifically, the records in the database will contain three pieces of information about cities: their populations, their economic "net worth" and their names. The population and the net worth are given as integers and the name is given as a string. You can compare the names using the comparison operators defined in the C++ string class. Your database will contain three separate 2-3 trees, storing records based on population, economic net worth, and city name, respectively.

Your database program should take inputs from **standard input** and write output to the **standard output**. **The name of the program should be "p2".**

The input will consist of a series of commands, one command per line. All commands are associated with a few parameters, separated by one or more whitespace characters. You do not need to check for syntax errors in the command lines. However, you do need to check for logical errors, such as inserting a record whose key is already in the tree. This will be described in detail later. Some of the commands will take a specific data called "field", with value as 1, 2, or 3. Field "1" is population, field "2" is economic net worth, and field "3" is name.

Commands, their syntax, their functions, and their corresponding outputs are as follows.

- **`insert`** *`<population> <economy> <name>`*

  Function: Insert a record into the database, with population as *`<population>`*,

economic net worth as <*economy*>, and name as <*name*>. You will create a new record object and insert it into each of the three 2-3 trees. Note that only records with each value of the three fields not duplicating the same field of those records already in the database will be inserted. Otherwise, the insertion attempt should be rejected. For example, suppose that the database contains only one record "1000 12000 Atown". Then, we are able to insert a new record "12000 1000 Btown". However, we are not able to insert a new record "1000 15000 Btown", since the first "population" field of the new record has a duplicated value 1000.

Output: When taking this command, you should first reprint the command:

insert <*population*> <*economy*> <*name*>

with a single space separating adjacent entries. Then, if the insertion succeeds, print:

```
Insertion successful
```

Otherwise, print:

```
Insertion failed
```

- **search** <*field*> <*value*>

  Function: Search the record with the value as <*value*> in the field <*field*>.

  Output: When taking this command, you should first reprint the command:

  search <*field*> <*value*>

  with a single space separating adjacent entries. If such a record exists, print:

  Record found: <*population*> <*economy*> <*name*>

  where <*population*>, <*economy*>, and <*name*> are the population, the economic net worth, and the name part of the record, respectively. Otherwise, print:

```
No record found
```

- **remove** <*field*> <*value*>

  Function: Remove the record with the value as <*value*> in the field <*field*> from the database, if it exists. To do so, **you will need to remove the record from all three 2-3 trees**. If the record does not exist, the removal attempt should be rejected.

  Output: When taking this command, you should first reprint the command:

  remove <*field*> <*value*>

  with a single space separating adjacent entries. If such a record exists, print:

  Remove record: <*population*> <*economy*> <*name*>

  where <*population*>, <*economy*>, and <*name*> are the population, the economic net worth, and the name part of the record to be removed, respectively. Otherwise, print:

  ```
  Removal failed
  ```

- **list** <*field*>

  Function: List out the records in the database, sorted by the values in the field <*field*>.

  Output: When taking this command, you should first reprint the command:

  list <*field*>

  with a single space separating adjacent entries. Then, you should list out the records in the database, sorted by the values in the field <*field*>. The listing should be processed by an in-order traversal of the appropriate tree, with each record printed on a separate line. If the record appears in a node at level *k* in the tree (assume the first level is level 0), it will be printed with 4*k* spaces at the beginning.

III. Program Arguments and Error Checking

Your program takes no arguments. You do not need to do any error checking. You can assume that all the inputs are syntactically correct.

IV. Implementation Requirements and Restrictions

- You must use your 2-3 tree ADT to implement the database application. You may use any type you see fit as the templated type.
- You must make sure that your code compiles successfully on a Linux operating system. We provide you with a sample `Makefile`. You should change it according to your source code files and submit it together with your source code files. Your `Makefile` should compile the database program and **the program should be named as `p2` exactly.**
- You should name C++ source code files with the extension ".C" (**CAPITAL C**) instead of ".cpp".
- You may not leak memory in any way. To help you check if you are leaking memory, you may wish to call `valgrind`, which can tell whether you have any memory leaks. (You need to install `valgrind` first if your system does not have this program.) The `Makefile` we supply contains a rule to call `valgrind`. The rule is:

  ```
  memcheck: $(TARGETS)
     valgrind --leak-check=full ./$(TARGETS) $(MEMCHECKARGS)
  ```

  You can check memory leak by typing "`make memcheck`" in your Linux terminal. However, you should set the macro `MEMCHECKARGS` as the arguments of the program you are testing. For example, if you want to check whether running program

  ```
  ./p2 < database.in
  ```

  causes memory leak, then you should set

  ```
  MEMCHECKARGS = < database.in
  ```

- You may `#include <iostream>`, `<fstream>`, `<sstream>`, `<string>`, `<cstdlib>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library, including standard template library (STL).
- Output should only be done where it is specified.

V. Source Code Files and Compiling

There are one header file `two-three-tree.h` and one sample `Makefile` located in the programming-project-two-related-files.zip from our Sakai Resources.

You should copy these files into your working directory. **DO NOT modify `two-three-tree.h`!** However, you should modify `Makefile` to compile the program `p2` and to check memory leaks for different test cases.

You need to implement all the methods of the 2-3 tree ADT together with the helper functions in a file called `two-three-tree.C`. When implementing database application, you can define your own source files.

In order to guarantee that the TAs compile and test your program successfully, you should make sure that the database program compiled using your `Makefile` is named as `p2`. For this project, the penalty for code that does not compile will be **severe**, regardless of the reason. We will test your implementation of the 2-3 tree methods by building a program from **our** `two-three-tree.h` (in order to make sure that you do not modify it), **your** `two-three-tree.C`, and **our** test file `test.C`. We will test your database program by building a program from **our** `two-three-tree.h`, **your** `two-three-tree.C`, and the other of **your** source code files using **your** `Makefile`.

VI. Testing

We provide you with a file called `test-tree.C` in the programming-project-two-related-files.zip to help you test a few methods of the 2-3 tree ADT. You should compile `test-tree.C` together with our `two-three-tree.h` and your `two-three-tree.C`. If it compiles successfully, type the following into the Linux terminal (assume the compiled program is called `test-tree`):

```
./test-tree > test.out
diff test.out test-tree.out
```

The file `test-tree.out` records the output by our program and is also located in the programming-project-two-related-files.zip. If the `diff` program reports any differences at all, you have a bug.

We have also supplied an input file called `database.in` for you to test your database program. To do this test, type the following into the Linux terminal once your program has been compiled:

```
./p2 < database.in > test.out
diff test.out database.out
```

The file `database.out` records the output by our program and is also located in the programming-project-two-related-files.zip. If the `diff` program reports any differences at all, you have a bug.

These are the minimal amount of tests you should run to check your program. Those programs that do not pass these tests are not likely to receive much credit. You should also write other different test cases yourself to test your program extensively.

You should also check whether there is any memory leak using `valgrind` as we discussed above. For those programs that behave correctly but have memory leaks, they only get half of the grade.

VII. Submitting and Due Date

You should submit a `two-three-tree.C`, a `Makefile`, and the other related source files via the Assignment tool on Sakai. The `Makefile` compiles a program named `p2` for the database program. The due date is 11:59 pm on Nov 30th, 2012.

VIII. Grading

Your program will be graded along four criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style
4. Performance

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. **For those programs that**

behave correctly but have memory leaks, they will only get half of the implementation points. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions. Part of your grade will also be determined by the performance of your algorithm. Algorithms that run slowly may receive only a portion of the performance points.