Project 1: It's Time for Dodger Baseball! (due Friday, September 28th, 11:59 PM)

Every year, there seems to be annual ritual. In the world, we've been told there are two things guaranteed in life: death and taxes. However, someone forgot to mention that this applies to another thing: changing players on teams in any sport. At Chavez Ravine, this just happened with the acquisitions of Manny Machado, Brian Dozier, Ryan Madson, and David Freese (among others). But, it's never just about the players coming. It's also about players that go as part of the trades (and their families) that get affected as well. Some get to stay (cue up "Elian can stay!"), while some have to go (cue up either "Bye, Felicia!" or as our President said on a TV show called The Apprentice, "You're fired!"). Just food for thought.

In this project, you will write the implementation of the BaseballRoster as a map with a variety of data structures at your disposal (which will be detailed at the end). You will also implement a couple of functions that will operate on this BaseballRoster.

Implement BaseballRoster

Consider the following BaseballRoster interface:

```
typedef std::string PType;
typedef int NType;

class BaseballRoster
{
  public:
    BaseballRoster();        // Create an empty BaseballRoster map

    bool noPlayers() const;  // Return true if the BaseballRoster map
                             // is empty, otherwise false.

    int numberOfPlayers() const;  // Return the number of players in
                                  // the BaseballRoster map.

    bool addPlayer(const PType& player, const NType& number);
      // If player is not equal to any player currently in the
      // BaseballRoster, and if the player/number pair can be added to
      // the BaseballRoster, then do so and return true. Otherwise,
      // make no change to the BaseballRoster and return false
      // (indicating that the player is already in the
      // BaseballRoster).

    bool updatePlayer(const PType& player, const NType& number);
      // If player is equal to a player currently in the
      // BaseballRoster, then make that player no longer map to the
      // number it currently maps to, but instead map to the number of
      // the second parameter; return true in this case. Otherwise,
      // make no change to the BaseballRoster and return false.
    bool addOrUpdate(const PType& player, const NType& number);
      // If player is equal to a player currently in the
      // BaseballRoster, then make that player no longer map to the
      // number it currently maps to, but instead map to the number of
      // the second parameter; return true in this case. If player is
      // not equal to any player currently in the BaseballRoster, then
      // add it and return true. In fact, this function always returns
      // true.

    bool dfa(const PType& player); // Designate for Assignment
      // If player is equal to a player currently in the
      // BaseballRoster, remove the player/number pair with that
      // player from the map and return true.  Otherwise, make no
      // change to the BaseballRoster and return false.

    bool playerOnRoster(const PType& player) const;
      // Return true if player is equal to a player currently in the
      // BaseballRoster, otherwise false.

    bool findPlayer(const PType& player, NType& number) const;
      // If player is equal to a player currently in the
      // BaseballRoster, set number to the number in the map that that
      // key maps to, and return true.  Otherwise, make no change to
      // the number parameter of this function and return false.

    bool findPlayer(int i, PType& player, NType& number) const;
      // If 0 <= i < numberOfPlayers(), copy into the player and
      // number parameters the player and number of one of the
      // player/number pairs in the map and return true.  Otherwise,
      // leave the player and number parameters unchanged and return
      // false. (See below for details about this function.)

    void swapRoster(BaseballRoster& other);
      // Exchange the contents of this map with the other one.
};
```

The three-argument `findPlayer()` function enables a client to iterate over all elements of a `BaseballRoster` because of this property it must have: If nothing is inserted into or erased from the map in the interim, then calling that version of `get numberOfPlayers()` times with the first parameter ranging over all the integers from 0 to `numberOfPlayers()` – 1 inclusive will copy into the other parameters every key/value pair from the map exactly once. The order in which player/numbers pairs are copied is up to you. In other words, this code fragment

```
    BaseballRoster b;
    b.addPlayer("A", 10);
    b.addPlayer("B", 44);
    b.addPlayer("C", 10);
    string all;
    int total = 0;
    for (int z = 0; z < b.numberOfPlayers(); z++)
    {
        string p;
        int n;
        b.findPlayer(z, p, n);
        all += p;
        total += n;
    }
    cout << all << total;
```

must result in the output being exactly one of the following: `ABC64`, `ACB64`, `BAC64`, `BCA64`, `CAB64`, or `CBA64`, and the client can't depend on it being any particular one of those six. If the `BaseballRoster` is modified between successive calls to the three-argument form of `findPlayer()`, all bets are off as to whether a particular key/value pair is returned exactly once.

If nothing is inserted into or erased from the map in the interim, then calling the three-argument form of get repeatedly with the same value for the first parameter each time must copy the same key into the second parameter each time and the same value into the third parameter each time, so that this code is fine:

```
    BaseballRoster pitchers;

    pitchers.addPlayer("Clayton Kershaw", 22);
    pitchers.addPlayer("Kenley Jansen", 74);
    int n;
    string p1;
    assert(pitchers.findPlayer(1,p1,n) && (p1 == "Clayton
Kershaw" || p1 == "Kenley Jansen"));
    string p2;
    assert(pitchers.findPlayer(1,p2,n) && p2 == p1);
```

Notice that the empty string is just as good a string as any other; you should not treat it in any special way:

```
    BaseballRoster hitters;

    hitters.addPlayer("Justin Turner", 10);
    assert(!hitters.playerOnRoster (""));
    hitters.addPlayer("Cody Bellinger", 35);
    hitters.addPlayer("", 0);
    hitters.addPlayer("Yasiel Puig", 66);
    assert(hitters.playerOnRoster("", ""));
    hitters.dfa("Yasiel Puig");
    assert(hitters.numberOfPlayers() == 3
           && hitters.playerOnRoster("Justin Turner")
           && hitters.playerOnRoster("Cody Bellinger")
           && hitters.playerOnRoster("", ""));
```

When comparing keys for `addPlayer`, `updatePlayer`, `addOrUpdate`, `dfa`, `playerOnRoster`, and `findPlayer`, just use the == or != operators provided for the string type by the library. These do case-sensitive comparisons, and that's fine.

For this project, implement this `BaseballRoster` interface using your choice of data structure (dynamically resizeable array, singly-linked list, doubly-linked list, tree, or hash table). (You must not use any contaner from the C++ library.)

For your implementation, if you let the compiler write the destructor, copy constructor, and assignment operator, they will do the wrong thing, so you will have to declare and implement these public member functions as well:

Destructor

When a `BaseballRoster` is destroyed, all dynamic memory must be deallocated.

Copy Constructor

When a brand new `BaseballRoster` is created as a copy of an existing `BaseballRoster`, a deep copy should be made.

Assignment Operator

When an existing `BaseballRoster` (the left-hand side) is assigned the value of another `BaseballRoster` (the right-hand side), the result must be that the left-hand side object is a duplicate of the right-hand side object, with no memory leak (i.e. no memory from the old value of the left-hand side should be still allocated yet inaccessible).

Notice that there is now no *a priori* limit on the maximum number of player/number pairs in the `BaseballRoster` (so `addOrUpdate` should always return true). Also, if a `BaseballRoster` has a size of n, then the values of the first parameter to the three-parameter form of `findPlayer` for which that function retrieves a player and a number and returns true are 0, 1, 2, ..., n-1; for other values, it returns false without setting its second and third parameters.

Another requirement is that the number of statement executions when swapping two `BaseballRoster`s must be the same no matter how many player/number pairs are in the `BaseballRoster`s.

Implement Some Non-Member Functions

Using only the *public* interface of `BaseballRoster`, implement the following two functions. (Notice that they are non-member functions; they are not members of `BaseballRoster` or any other class.)

```
bool combineRosters(const BaseballRoster& brOne,
                    const BaseballRoster& brTwo,
                    BaseballRoster& brMerged);
```

When this function returns, `brMerged` must consist of pairs determined by these rules:

If a full name appears in exactly one of `brOne` and `brTwo`, then `brMerged` must contain an element consisting of that full name and its corresponding value.

If a full name appears in both `brOne` and `brTwo`, with the same corresponding value in both, then `brMerged` must contain an element with that full name and value.

When this function returns, `brMerged` must contain no elements other than those required by these rules. (You must not assume `brMerged` is empty when it is passed in to this function; it might not be.)

If there exists a full name that appears in both `brOne` and `brTwo`, but with different corresponding values, then this function returns false; if there is no full name like this, the function returns true. Even if the function returns false, result must be constituted as defined by the above rules.

For example, suppose a `BaseballRoster` maps the full name to integers. If `brOne` consists of these three elements

```
 "Ross Stripling" 68   "Walker Buehler" 21   "Kenta Maeda" 18
```

and `brTwo` consists of

```
 "Kenta Maeda" 18   "Alex Wood" 57
```

then no matter what value it had before, `brMerged` must end up as a list consisting of

```
 "Ross Stripling" 68   "Walker Buehler" 21   "Kenta Maeda" 18
 "Alex Wood" 57
```

and `combineRosters` must return true.

If instead, `brOne` consists of

```
 "Brian Dozier" 6   "David Freese" 25   "Manny Machado" 8
```

and `brTwo` consists of

```
 "Manny Machado" 13   "Ryan Madson" 50
```

then no matter what value it had before, `brMerged` must end up as a list consisting of

```
 "Brian Dozier" 6   "David Freese" 25   "Ryan Madson" 50
```

and `combineRosters` must return false.

```
void outright(const BaseballRoster& brOne,
              const BaseballRoster& brTwo,
              BaseballRoster& brResult);
```

When this function returns, `brResult` must contain a copy of all the pairs in `brOne` whose players don't appear in `brTwo`; it must not contain any other pairs. (You must not assume result is empty when it is passed in to this function; it may not be.)

For example, if `brOne` consists of the three pairs (in any order)

```
 "Don Mattingly" 8   "Dave Roberts" 30   "Joe Torre" 6
```

and if `brTwo` consists of the three pairs (in any order)

```
 "Tommy Lasorda" 2   "Don Mattingly" 8   "Joe Torre" 6
```

then no matter what value it had before, `brResult` must end up as a `BaseballRoster` consisting of

```
 "Dave Roberts" 30   "Tommy Lasorda" 2
```

Be sure these functions behave correctly in the face of aliasing: What if `brOne` and `brResult` refer to the same `BaseballRoster`, for example?

<u>**Other Requirements**</u>

Regardless of how much work you put into the assignment, your program will receive a low score for correctness if you violate these requirements:

- Your class definition, declarations for the two required non-member functions, and the implementations of any functions you choose to inline must be in a file named `BaseballRoster.h`, which must have appropriate include guards. The implementations of the functions you declared in `BaseballRoster.h` that you did not inline must be in a file named `BaseballRoster.cpp`. Neither of those files may have a main routine (unless it's commented out). You may use a separate file for the main routine to test your `BaseballRoster` class; you won't turn in that separate file.

- Except to add a destructor, copy constructor, assignment operator, and dump function (described below), you must not add functions to, delete functions from, or change the public interface of the `BaseballRoster` class. You must not declare any additional struct/class outside the `BaseballRoster` class, and you must not declare any public struct/class inside the `BaseballRoster` class. You may add whatever private data members and private member functions you like, and you may declare private structs/classes inside the `BaseballRoster` class if you like. The source files you submit for this project must not contain the word friend. You must not use any global variables whose values may be changed during execution.
- If you wish, you may add a public member function with the signature `void dump() const`. The intent of this function is that for your own testing purposes, you can call it to print information about the map; we will never call it. You do not have to add this function if you don't want to, but if you do add it, it must not make any changes to the map; if we were to replace your implementation of this function with one that simply returned immediately, your code must still work correctly. The dump function must not write to `cout`, but it's allowed to write to `cerr`.
- Your code must build successfully (under both Visual C++ and either clang++ or g++) if linked with a file that contains a main routine.
- You must have an implementation for every member function of `BaseballRoster`, as well as the non-member functions `combineRosters` and `outright`. Even if you can't get a function implemented correctly, it must have an implementation that at least builds successfully. For example, if you don't have time to correctly implement `BaseballRoster::dfa` or `outright`, say, here are implementations that meet this requirement in that they at least build successfully:

```
bool BaseballRoster::dfa(const PType& player)
{
   return false; // not correct, but at least this code compiles
}

void outright(const BaseballRoster& brOne, const BaseballRoster&
   brTwo, BaseballRoster& brResult) {
   return; // not correct, but at least this code compiles
}
```

You've probably met this requirement if the following file compiles and links with your code. (This uses magic beyond the scope of CS 30.)

```
#include "BaseballRoster.h"
#include <type_traits>

#define CHECKTYPE(f, t) { auto p = (t)(f); (void)p; }

static_assert(std::is_default_constructible<BaseballRoster>::val
ue, "BaseballRoster must be default-constructible.");

static_assert(std::is_copy_constructible<BaseballRoster>::value,
                "BaseballRoster must be copy-constructible.");

void ThisFunctionWillNeverBeCalled()
{
   CHECKTYPE(&BaseballRoster::operator=, BaseballRoster&
      (BaseballRoster::*)(const BaseballRoster&));
   CHECKTYPE(&BaseballRoster::noPlayers, bool(BaseballRoster::*)
      () const);
   CHECKTYPE(&BaseballRoster::numberOfPlayers, int
      (BaseballRoster::*)() const);
   CHECKTYPE(&BaseballRoster::addPlayer, bool(BaseballRoster::*)
      (const PType&, const NType&));
   CHECKTYPE(&BaseballRoster::updatePlayer, bool
      (BaseballRoster::*)(const PType&, const NType&));
   CHECKTYPE(&BaseballRoster::addOrUpdate, bool
      (BaseballRoster::*)(const PType&, const NType&));
   CHECKTYPE(&BaseballRoster::dfa, bool (BaseballRoster::*)
      (const PType&));
   CHECKTYPE(&BaseballRoster::playerOnRoster, bool
      (BaseballRoster::*)(const PType&) const);
   CHECKTYPE(&BaseballRoster::findPlayer, bool
      (BaseballRoster::*)(const PType&, NType&) const);
   CHECKTYPE(&BaseballRoster:: findPlayer, bool
      (BaseballRoster::*)(int, PType&, NType&) const);
   CHECKTYPE(&BaseballRoster::swapRoster, void
      (BaseballRoster::*)(BaseballRoster&));

   CHECKTYPE(combineRosters,  bool (*)(const BaseballRoster&,
      const BaseballRoster&, BaseballRoster&));
   CHECKTYPE(outright, void (*)(const BaseballRoster&,
      const BaseballRoster&, BaseballRoster&));
}
```

```
int main()
{}
```

If you add `#include <string>` to `BaseballRoster.h`, have the `typedef` define `NType` as `std::string`, and link your code to a file containing

```
#include "BaseballRoster.h"
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

void test()
{
   BaseballRoster bench;

   assert(bench.addPlayer("Chase Utley", "Silver Fox"));
   assert(bench.addPlayer("Rich Hill", "D. Mountain"));
   assert(bench.numberOfPlayers() == 2);

   string original = "Chicken Strip";
   assert(bench.findPlayer("Chase Utley", original)  &&
      original == "Silver Fox");
   original = "Sam";
   string s1;
   assert(bench.findPlayer(0, s1, original)
      && ((s1 == "Chase Utley" && original == "Silver Fox")
         || (s1 == "Rich Hill" && original == "D. Mountain")));
   original = "Yazmanian Devil";
   string s2;
   assert(bench.findPlayer(1, s2, original)  &&  s1 != s2  &&
      && ((s2 == "Chase Utley" && original == "Silver Fox")
         || (s2 == "Rich Hill" && original == "D. Mountain")));

   return;
}

int main()
{
   test();
   cout << "Passed all tests" << endl;
   return 0;
}
```

the linking must succeed. When the resulting executable is run, it must write `Passed all tests` to `cout` and nothing else to `cout`.

If we successfully do the above, then make no changes to `BaseballRoster.h` other than to change the typedefs for `BaseballRoster` so that `PType` specifies `int` and `NType` specifies `std::string`, recompile `BaseballRoster.cpp`, and link it to a file containing

```
#include " BaseballRoster.h"
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

void test()
{
   BaseballRoster bullpen;

   assert(bullpen.addPlayer(52, "Pedro Baez"));
   assert(bullpen.addPlayer(75, "Scott Alexander"));
   assert(bullpen.numberOfPlayers() == 2);
   string name;
   assert(bullpen.findPlayer(52, name) && name == "Pedro Baez");
   name = "";
   int i1;
   assert(bullpen.findPlayer(0, i1, name)
      && ((i1 == 52 && name == "Pedro Baez")
         || (i1 == 75 && name == "Scott Alexander")));
   int i2;
   name = "";
   assert(bullpen.findPlayer(1, i2, name) && i1 != i2
      && ((i2 == 52 && name == "Pedro Baez")
         || (i2 == 75 && name == "Scott Alexander")));

   return;
}

int main()
{
   test();
   cout << "Passed all tests" << endl;
   return 0;
}
```

the linking must succeed. When the resulting executable is run, it must write `Passed all tests` to `cout` and nothing else to `cout`.

During execution, if a client performs actions whose behavior is defined by this spec, your program must not perform any undefined actions, such as dereferencing a null or uninitialized pointer.

Your code in `BaseballRoster.h` and `BaseballRoster.cpp` must not read anything from `cin` and must not write anything whatsoever to `cout`. If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. `cerr` is the standard error destination; items written to it by default go to the screen. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

If you decide to implement the `BaseballRoster` using a hash table you must also declare and define the following two functions in `BaseballRoster.cpp`. If a `PType` is specified by a `std::string` it will call the first function, if it is specified by a `int` it should call the second function.

```
int Convert_Key(std::string key)
{
    // your code for hashing a string goes here
}

int Convert_Key(int key)
{
    // your code for hashing a int goes here
}
```

Grading

Here is the grading scale for the amount you can earn on this project for picking a data structure.

Data Structure	**Possible Points**
Dynamically resizable array	95 (out of 100)
Linked list (singly)	100
Linked list (doubly)	105 (up to 5 extra credit points)
Binary tree	105 (up to 5 extra credit points)
Hash table	110 (up to 10 extra credit points)

Turn It In

There will be a link on Canvas that will enable you to turn in your source files and report. You will turn in a zip file containing these files:

- `BaseballRoster.h`. When you turn in this file, the typedefs must specify `std::string` as the `PType` and `int` as the `NType`.
- `BaseballRoster.cpp`. Function implementations should be appropriately commented to guide a reader of the code.
- A file named `report.doc` or `report.docx` (in Microsoft Word format) or `report.txt` (an ordinary text file) that contains:
    - A description of the design of your implementation and why you chose it. (A couple of sentences will probably suffice, perhaps with a picture of a typical Map and an empty Map. Is your list circular? Does it have a dummy node? What's in your nodes?)
    - A brief description of notable obstacles you overcame.
    - Pseudocode for non-trivial algorithms (e.g., `BaseballRoster::dfa` and `outright`)
    - A list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. For example, here's the beginning of a presentation in the form of code:

The tests were performed on a map from strings to doubles

```
  // default constructor
BaseballRoster br;
  // For an empty list:
assert(br.numberOfPlayers() == 0); // test size
assert(br.noPlayers());            // test empty
assert(!br.dfa("Pat Venditte"));   // nothing to erase
```

Even if you do not correctly implement all the functions, you must still list test cases that would test them. Don't lose points by thinking "Well, I didn't implement this function, so I won't bother saying how I would have tested it if I had implemented it."