Assignment 6

Instructions: Write a makefile that has targets: prod, debug, test and clear, where prod builds your solution to problem A without debugging code, debug builds your solution to problem A with debugging code, test builds your unit test driver, and clear deletes any executable and object files you create for any of the targets. For problem A, write as many source files as you like and name them what you like, but make sure that the makefile works correctly. Tar and zip your C++ and make files and submit them via the turnin program.

A. Create a Hashtable template class. Your template class should take three template parameters, the first parameter should tell you what type of elements are in the hashtable, the second parameter is a function pointer to a hash function, and the third parameter should give the initial size of the table, with a default of 10. A hash function must take an object of the type used in the hashtable and return an unsigned int. The maximum load factor of your hashtable should be 75%. Your hashtable should not have a minimum load factor. Your hashtable should support the following member functions:

- void add(T): adds an element to the hashtable, if it is not already present. This must complete in constant time.
- bool delete(T): deletes an element from the hashtable, if it is present. Returns true if the element was successfully deleted, false if the hashtable did not contain the element.
- bool test(T): returns true if the element is present in the hashtable, false otherwise.
- iterator<T> iterator(): produces an iterator over the elements in the hashtable. This must return in constant time. In the presence of concurrent modification, the iterator returned should return NULL values on all subsequent calls.
- int size(): returns the size of the backing array (see below).
- int load(): returns the number of elements currently in the hashtable.

Create Unit Testing Driver for your hashtable class that creates and tests a hashtable for ints, C++ string objects, and C strings. Your test code should intersperse inserts and deletes, checking on the contents of the hashtable each time. Your test code should insert about 100 values of each type, and delete about 25 values.

Create a program using your hashtable, that lets me choose a fantasy football team. The team can have any number of players. For each player, I want to input a jersey number, name, whether the player is offensive, defensive, or special teams and I want to input the player’s position. I want to be able to add players, delete players and print the team roster. When I delete players, I should just be able to give the jersey number, but since
players come from different teams, there may be several players who have share a jersey number. In that case, your program should ask me which one I meant.
**Hashtables**

There are several options for implementing a hashtable. Here, we fully explain one method of implementing a hashtable.

At its simplest, a hashmap consisted of an array. When an item is inserted into the hashtable, the item’s hash is computed and moded with the size of the array. So, if the item’s hash is computed at 12746 and the array has 100 slots, the item would be placed in slot 46. When we want to know if the item is currently in the hashtable, we look in slot 46. However, suppose the next item hashes to 6546, then it would also belong in slot 46. This is referred to as a hash collision. To avoid hash collisions, the slots in the array hold pointers to linked lists of the elements that belong in the slot. In the case of insertions in the face of hash collisions, the appropriate linked list is extended. For each lookup, the linked list is searched until the end is reached or the item is found.

**Constant Time Choice Functionality**

To our hashtable, we also wish to add choice functionality that allows us to pick an element out of the hashtable in constant time (not possible if we have to search the slots for one that is non-empty). To this end, our backing array must be an array of pointers to holders for the hashtable values. Each holder consists of:

- The value: the item inserted into the hashtable
- The global next value: a pointer to a holder of an item in the hashtable or NULL
- The global previous value: a pointer to a holder of an item in the hashtable or NULL

The global pointers form a doubly linked list of all the items in the hashtable, most recently added first. The hashtable object has a copy of the first holder in the linked list, and whenever an item is inserted, the hashtable pointer is updated to point to the new item, which is inserted into the beginning of the doubly linked list. If we want to choose an element from the array, we can use the first pointer in the hashtable object. If we want a total order of the elements in the hashtable, we can use the global next pointer linked list.

**Concurrent Modification Protection**

We also want to add functionality that protects our iterators from concurrent modification. To accomplish this, the hashtable object will maintain a time value. Each time an insertion or deletion occurs, the time value should be incremented. Each iterator should also know what the time value was when it was created. If the hashtables current time value does not match the iterator’s time creation value, then the iterator should return NULL’s.

This protects the iterators from modification of the hash table within the current thread. To protect them from true concurrency, we would need to use mutex’s to ensure exclusive use of the hashtable by competing threads. We will not implement this in this assignment.
Hash Functions

The key to a good hash implementation is the hash function. There must be a different hash function for each type that may be used in a hash table. The hash function must be consistent with 
. That is, for elements and , if then ; however, it is not true that implies .

For example, the hash function for int might be

```c
unsigned int hashint(int i){
    if (i < 0) i = -i;
    return i;
}
```

The hash function for a long, might look like

```c
unsigned int hashlong(long l){
    unsigned int i = ~0;
    return l & i;
}
```

A hash function for a string might be

```c
unsigned int hashc_string(const char* s){
    unsigned int result = 0;
    while (*s != 0)
        result = ++result**s++;
    return result;
}
```

Notice that these are good hash functions because they distribute evenly through the integers. If the hashtable is accepting a type , then the function pointer definition for a hash function would be

```c
unsigned int (*hash)(T);
```

Rehashing

The load factor for a hashtable is the number of items currently in the hashtable divided by the number of slots. In general, hashtables have minimum and maximum allowable hashtables. When the hashtable reaches the maximum load factor, the backing array is replaced by a bigger one, all of the entries are copied from the old backing array and inserted into the new array. To give amortized constant time even in the face of rehashing, the following algorithm is used:

1. The hashtable always has two backing arrays, the current array and the old array.
2. New values are always inserted into the current array.
3. Whenever a value is inserted into the current array, a value in the old array is selected using the choice functionality, it is deleted and rehashed into the current array.
4. When values must be looked up, the old array is searched first, if it contains any values. Any values encountered in the old array are deleted and rehashed into the current array.
5. When the load factor of the current array would exceed the max load factor, the old array is guaranteed to be empty. The old array is deleted from the heap. The current array becomes the old array, and a new current array is created that is twice the size of the old one.

Iterators

For purposes of this assignment, the iterator returned by the iterator() function should be an input iterator. This means the following need to be functional, given an iterator `iter`:

- A copy constructor that takes a reference to another iterator.
- Overloaded `operator=` function to copy another iterator.
- Overloaded both `operator++` functions to change meaning of current value.
- Overloaded `operator==` and `operator!=` to indicate whether two iterators are from the same hashtable, have the same time state, and have the same current value.
- Overloaded `operator*` to return current value.

Pictorially, the following diagram shows the heap with a particular configuration of a hashtable.
After an insert operation (and the resulting move of any item from the old array to the current array), the heap may look like this.

The green items represent change. Since the hashtable’s current_time value has been updated, the iterator is no longer valid. When the iterator is next used, it will detect this situation and delete the red arrow.