# Homework – Closed Hashing (Hash Tables with Arrays)
## Also known as "The Monster Hash"

**Brad Peterson – Weber State University**

This assignment implements an array-based hash table. Your responsibility is to support:

- Arrays utilizing unique pointer arrays. No linked lists allowed.
- Keys of type string, values that are templated.
- The normal create/retrieve/exists/remove methods.
- Resizing of the internal arrays when the hash table hits 75% capacity.
- Iterators for the container.

The HashTable class is as follows:

```
//******************
//The HashTable class
//******************
template <typename T>
class HashTable
{
  friend class Iterator<T>;
public:
  unsigned int getNumBuckets() { return capacity; }
  unsigned int getTotalCount() const;
  unsigned int getWorstClump() const;

  // TODO: Create the two constructors
  //HashTable();
  //HashTable(const unsigned int capacity);

  // TODO: Add an operator= move method
  //HashTable<T>& operator=(HashTable<T>&& objToMove) noexcept;

  // We don't want the move constructor, copy constructor, or copy assignment.
  HashTable(HashTable<T>&& obj) = delete;
  HashTable(const HashTable<T>& obj) {
    cout << "Failed homework issue:  You hit the HashTable copy constructor.  That's bad!" << endl;
  }
  HashTable<T>& operator=(const HashTable& obj) {
    cout << "Failed homework issue:  You hit the HashTable copy assignment.  That's bad!" << endl;
    HashTable temp;
    return temp;
  }

  // TODO: supply these methods
  void create(const string& key, const T& item); // method for L - values
  //void create(const string& key, T&& item); // method for R - values
  //T retrieve(const string& key); //return by value, acts as a read only retrieve)
  //T& operator[](const string& key); // return by reference which can allow for modification of values
  //bool exists(const string& key); // return a true or false if a key is valid in the hash table
  //void remove(const string& key); // remove an element given a key
  //Iterator<T> begin();
  //Iterator<T> end();
private:
  unsigned int hash(const string& key) const;
  unsigned int capacity{ 20 };
  unsigned int count{ 0 };
  unique_ptr<int[]> status_arr;
  unique_ptr<string[]> key_arr;
  unique_ptr<T[]> value_arr;
}; // end class HashTable
```

This hash table uses an array-based design with arrays for status, key, and value. The status array holds a 0 if the slot is empty and has never been used, a 1 if the slot is currently in use, and a -1 if the slot was in use but removed. Both the key and value arrays have been merged into a `std::pair` array. The key value array thus becomes an array of `std::pair` objects. Each pair object has a key/first of type string, and a value/second of type T. (Note, you will never need to create a pair object one at a time. Your constructor will create all pairs as part of the array allocation. Your methods will only require you use a `std::pair`'s `.first` and `.second` data members.)

The hashing algorithm has been given to you, and you need only call `hash(key)`, and it returns the bucket index:

```
template <typename T>
unsigned int HashTable<T>::hash(const string& key) const {

  return std::hash<std::string>{}(key) % capacity;

}
```

## Implement the following:

- **The public default constructor:** Allocate the two arrays. Note that these are unique pointers. You need to use:

  ```
  this->status_arr = make_unique<int[]>(capacity);
  this->keyValue_arr = make_unique<something else[]>(capacity);
  ```

  Note that the key value array holds pairs of strings and T data.

  Set all status array elements to zero using a simple for loop.

- **The custom capacity constructor**: Similar to the default public constructor. This constructor accepts a defined capacity. Create the arrays to hold that many items, set all values of the status array to zero, and set the capacity to the parameter value.

- **The L-Value create() method (the one with a const T& item):** Do an initial check if the count + 1.0 is 75% of the capacity. (The code must do `count + 1.0` to make it double math and work. If the code did `count + 1`, that is integer math, and an integer divided by a larger integer capacity is always integer 0, which not what you want. For example, if capacity is 80 and count is 59, then (59+1)/80 = 0 in integer math, but (59+1.0)/80 = .75 in double math.)

  If it is over capacity, trigger a rehash.

  (Note, in order to work on the assignment one unit test at a time, avoid implementing this rehash code logic until you arrive at the rehash tests, and until that time just skip to "The rest of the create method", and come back to the rehash code much later when you arrive at those tests.)

  - **Rehash:**
    - In your `create()` method's rehash logic, create a new hash table object, size it to the current capacity * 2. In other words, create a HashTable object invoking that private constructor just implemented.
    - Write a loop that iterates capacity times.
      - If the status array is 1 at this particular index, then call the new hash table object's create method. For the first argument, pass in the old hash table's key at this index. For the second argument, pass in an std::move of the old hash table's value at this index. We don't move the key, but we do move the value. We don't need to copy in the status, as the new hash table manages its own hashing, placing, and setting the status array of its own internal arrays.

        - When the loop is finished, move the new hash table object back into `*this` with `*this = std::move(newHashTableObject);`. Note that this requires that `operator=` move assignment be implemented.
    - **The rest of the create method:**
        - Hash they key and obtain the result, with `unsigned int index = hash(key)`. That result is your starting index. You do not need to hash the key again in this method, as you have your starting point. Set up an infinite loop (such as `while(true)`) which searches starting at index for the first status array slot whose value is 0 or -1.
            - When you find an open slot, insert the key and value array into the keyValue array, using `.first` and `.second` respectively. For example, `keyValue_arr[someIndex].first = key;`, which stores the key. Do something similar to store the value. Update the status array at that index to 1. Increment count.
            - When looking for an open slot, make sure you allow the index to wrap back to 0 if needed. This will require two indexes. One which simply ensures a loop iterates at most capacity times. Another which starts at the hash index, and possibly wraps back around to 0.

- **The R-Value create() method (the one with a T&& item):** This has the exact same logic as the L-Value create() method described previously, with one exception: this method requires using `std::move(item_to_move)` in two spots. Thus, if you have the L-value create() method working, you should copy and paste its code into the R-value create method. Then you should make two adjustments. First, in the rehash logic, when the new hash table's create() is invoked, for the second argument, change it so that the second argument is wrapped with std::move(). Second, in the actual assignment, use `= std::move(item_to_move)` instead of plain copy `=` when assigning the value into the value array.

- **The exists() method:** The code needs to hash the key obtaining an index. Then starting at that index, search for status array elements that are 1 or -1. If it's a 1, see if the key matches. If so, return true. If not, keep searching. If the status array is 0, then the item doesn't exist, so return false. If while searching the index is past the bounds of the array, index should wrap back to zero. It should keep searching until it has searched every possible element. If after looping nothing was found, return false.

- **The remove() method**: This is largely the same as the exists() method. The difference being that instead of returning true, you instead set the status array to -1 and return. Decrement count.

- **The retrieve() method**: This is largely the same as the exists() method. The difference being that instead of returning true, you instead return the value at that index and return. If you don't find it, throw a std::logic_error.

- **The operator[]() method**: This is largely the same as the retrieve() method, it just doesn't allow for a copy of the data anywhere.

- **The move assignment (operator=):** First check if the address of the object to be moved is different than this. If so, proceed.

  Move these things: the status array, and the key value array, into their corresponding "this" object. For example, to move the status array:

  ```
  this->status_arr = std::move(obj.status_arr);
  ```

  Copy the capacity and count from the parameter object into the "this" object. Set the parameter object's capacity and count to zero. (You don't move ints, you just make a copy into the destination and then zero out the source.) Finish the method with `return *this;`

**The Iterator class is as follows:**

```
//************************************************************
//The Iterator class
//************************************************************
template <typename T>
class Iterator {
  friend class HashTable<T>;
public:

  // TODO: define these methods
  //pair<string, T>& operator*();
  //bool operator!=(const Iterator<T>& rhs) const;
  //Iterator<T> operator++();
private:
  HashTable<T>* hashTable{ nullptr };
  unsigned int index{ 0 };
};
```

The design of this Iterator uses two private data members. The first retains a pointer to the iterator object, so that you can then go into that object and obtain all private data members. The second retains which array index it is currenting "pointing at".

- **HashTable's begin() method:** Create an iterator object. Set its hashTable data member to this (so it has an address to the hash table object). Create a loop that iterates at most capacity times. Look for the first status array element that is 1. Set the iterator object's index value to that index. If no status array element is found with a 1, set the iterator object's index value to capacity. Return the iterator object.

- **HashTable's end() method:** Create an iterator object. Set its hashTable data member to this (so it has an address to the hash table object). Set the iterator object's index value to capacity. Return the iterator object.

- **Iterator's operator*() method:** Simply return the hashTable's key array at the index.

- **Iterator's operator!=() method:** Compare two things: 1) both iterator's hashTable pointer address. 2) both iterator's index value.

- **Iterator's operator++() method:** Use a loop that iterates up to the hash table's capacity. Increment the index value. If the hash table's status array at the index value is 1, `return *this`. Otherwise, iterate again.