# Homework 5 – Closed Hashing (Hash Tables with Arrays)
## Also known as "The Monster Hash"

**Brad Peterson – Weber State University**

For this assignment, we want to implement an array-based hash table. In this assignment, the key data type and the value data type will be a template. Your responsibility is to support:

- Arrays utilizing unique pointer arrays. No linked lists allowed.
- 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 array
  //HashTable();
  // TODO: Add an operator= move method
  //HashTable<T>& operator=(HashTable<T>&& obj) noexcept;

  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;
  }
  // We won't implement this move constructor for the homework
  HashTable(HashTable<T>&& obj) = delete;

  // 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); // method(return by value, acts as a read only retrieve)
  //T& operator[](const string& key); // method(return by reference which can allow for modification of
values)
  //bool exists(const string& key);//  method(returns a boolean)
  //void remove(const string& key); //method
  //Iterator<T> begin();
  //Iterator<T> end();
private:
  // TODO: Implement a private constructor that sizes the arrays to customCapacity
  //HashTable(const unsigned int customCapacity);
  unsigned int hash(const string& key) const;
  unsigned int capacity{ 20 };
  unsigned int count{ 0 };
  unique_ptr<int[]> status_arr;
  unique_ptr<pair<string, T>[]> keyValue_arr;
};// end class HashTable
```

- **The public default constructor:** Allocate the status array and the key value array to hold capacity items. (The initial capacity is 20). Perform a loop to set all status_arr elements to zero.
- **The L-Value create() (the one with a const T& item) and R-Value method (the one with a T&& item):** Do an initial check if the count + 1.0 is 75% of the capacity (be very careful and make sure it does proper division using doubles and not integer division). If it is over capacity, trigger a rehash. (Note, don't code in this rehash until you hit the rehash tests.):
  - **Rehash:**
    - Implement the private hashTable constructor which sizes the arrays to the value found in the parameter.
    - In your create() method's rehash logic, create a new hash table object, size it to the current capacity * 2.
    - Write a loop that iterates capacity times
      - If the status array is 1 at this particular index, then call the new hash table's create method. Pass in the old hash table's key at this index and move in the old hash table's value at this index. (The goal is to let the new hash table manage its own hashing and its own placing into its own arrays)
    - When the loop is finished, move the new hash table object back into *this.
  - **The rest of the create method:**
    - Starting at the index indicated by the hash value, look for the first open slot (a slot that is 0 or -1). When you find an open slot, insert the data on into the keyValue_arr. Note that the key is in the pair's first, and the value is in the pair's second.
    - When looking for an open slot, make sure you allow the index to wrap back to 0 if needed. Tthis 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 aback around to 0.
- **The exists() method:** The code needs to hash the key obtaining an index. Then starting at that index, search for statusArray 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 statusArray 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**: it is largely the same as the exists() method. The difference being that instead of returning true, you instead set the statusArray to -1 and return.
- **The retrieve() method**: it 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 1.
- **The operator[]() method**: it is largely the same as the retrieve() method, it just doesn't allow for a copy of the data anywhere.
- **The move assignment (operator=):** Move these things: the key value array and the status array into the corresponding data members in "this" object. Copy the capacity and count from the parameter object into the corresponding data members "this" object. Set the parameter object's capacity and count to zero. Finish with a "return *this;"
- **The private constructor**: Similar to the default public constructor. The private 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 Iterator class is as follows:**

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 };
};
```

- **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 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 keyValue_arr at the index. The keyValue_arr holds pair objects, and this returns the tpair object.
- **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 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.