My code is bolded, also sorry about previous formatting issue I hope this works better for you!

```
#include "HashTable.h"
#include <iomanip>  // for use of setw
#include <cmath>
using namespace std;

// a new hash table whose capacity is the prime number closest to
// and greater that 2 times the capacity of the old hash table (use next_prime)
// replaces the old hash table and all items from the old hash table
// are rehashed (re-inserted) into the new hash table
// (the old hash table is discarded - memory returned to heap)
// (HINT: put next_prime and insert to good use)
void HashTable::rehash() {
  string tempArr[used];
  int counter = 0;
  for (size_type i = 0; i < capacity; i++) {
    if (data[i].word != "") tempArr[counter++] = data[i].word;
    if (counter == used) break;
  }
  size_type newCapacity = next_prime(capacity * 2);
  Item *temp = new Item[newCapacity];
  delete [] data;
  data = temp;
  used = 0;
  capacity = newCapacity;
  for (size_type i = 0; i < counter; i++) insert(tempArr[i]);
}

// returns true if sWord already exists in the hash table,
```

```
// otherwise returns false
bool HashTable::exists(const string& sWord) const
{
  for (size_type i = 0; i < capacity; ++i)
    if ( data[i].word == sWord ) return true;
  return false;
}

// returns true if sWord can be found in the hash table
// (MUST use hashing technique, NOT doing a linear search
// like what is done in exists above),
// otherwise return false
// CAUTION: major penalty if not using hashing technique
bool HashTable::search(const string& sWord) const {
  size_type loc1, loc0, key = hash(sWord);
  loc1 = loc0 = key % capacity;
  for (int i = 1; i < capacity; i++) {
    if (data[loc1].word == sWord) return true;
    if (data[loc1].word == "") return false;
    loc1 = (loc0 + i^2) % capacity;
  }
  return false;
}

// returns load-factor calculated as a fraction
double HashTable::load_factor() const
{ return double(used) / capacity; }

// returns hash value computed using the djb2 hash algorithm
```

```
// (2nd page of Lecture Note 324s02AdditionalNotesOnHashFunctions)
HashTable::size_type HashTable::hash(const string& word) const {
  size_type hash = 5381;
  int ascii, i;
  for (i = 0; i < word.size(); i++) {
    ascii = word[i];
    hash = (hash * 33) + ascii;
  }
  return hash;
}

// constructs an empty initial hash table
HashTable::HashTable(size_type initial_capacity)
        : capacity(initial_capacity), used(0)
{
  if (capacity < 11)
    capacity = next_prime(INIT_CAP);
  else if ( ! is_prime(capacity))
    capacity = next_prime(capacity);
  data = new Item[capacity];
  for (size_type i = 0; i < capacity; ++i)
    data[i].word = "";
}

// returns dynamic memory used by the hash table to heap
HashTable::~HashTable() { delete [] data; }

// returns the hash table's current capacity
HashTable::size_type HashTable::cap() const
```

```
{ return capacity; }

// returns the # of hash-table slots currently in use (non-vacant)
HashTable::size_type HashTable::size() const
{ return used; }

// graphs a horizontal histogram that gives a decent idea of how
// items are distributed over the hash table
void HashTable::scat_plot(ostream& out) const
{
  out << endl << "Scatter plot of where hash table is used:";
  size_type lo_index = 0,
            hi_index = capacity - 1,
            width;
  if (capacity >= 100000)
    width = capacity / 250;
  else if (capacity >= 10000)
    width = capacity / 25;
  else
    width = capacity / 10;
  size_type max_digits = size_type( floor( log10(hi_index) ) + 1 ),
            label_beg  = lo_index,
            label_end  = label_beg + width - 1;
  for(label_beg = lo_index; label_beg <= hi_index; label_beg += width)
  {
    out << endl;
    if( label_end > hi_index)
      out << setw(max_digits) << label_beg << " - " << setw(max_digits) << hi_index << ": ";
    else
```

```
      out << setw(max_digits) << label_beg << " - " << setw(max_digits) << label_end << ": ";
    size_type i = label_beg;
    while ( i <= label_end && i <= hi_index)
    {
      if ( ! data[i].word.empty() )
        out << '*';
      ++i;
    }
    label_end = label_end + width;
  }
  out << endl << endl;
}

// dumping to out contents of "segment of slots" of the hash table
void HashTable::grading_helper_print(ostream& out) const
{
  out << endl << "Content of selected hash table segment:\n";
  for (size_type i = 10; i < 30; ++i)
    out << '[' << i << "]: " << data[i].word << endl;
}

// sWord (assumed to be currently non-existant in the hash table)
// is inserted into the hash table, using the djb2 hash function
// and quadratic probing for collision resolution
// (if the insertion results in the load-factor exceeding 0.45,
// rehash is called to bring down the load-factor)
void HashTable::insert(const string& sWord) {
  size_type loc1, loc0, key = hash(sWord);
  loc1 = loc0 = key % capacity;
```

```
  for (int i = 1; i < capacity; i++) {
    if (data[loc1].word == "") {
      data[loc1].word = sWord;
      used++;
      break;
    }
    loc1 = (loc0 + i^2) % capacity;
  }
  if (load_factor() > 0.45) rehash();
}

// adaptation of : http://stackoverflow.com/questions/4475996
//              (Howard Hinnant, Implementation 5)
// returns true if a given non-negative # is prime
// otherwise returns false
// making use of following:
//   if a # is not divisible by 2 or by 3, then
//   it is of the form 6k+1 or of the form 6k+5
bool is_prime(HashTable::size_type x)
{
  if (x <= 3 || x == 5) return true;
  if (x == 4 || x == 6) return false;

  HashTable::size_type inc = 4;
  for (HashTable::size_type i = 5; true; i += inc)
  {
    HashTable::size_type q = x / i;
    if (q < i)
      return true;
```

```
    if (x == q * i)
      return false;
    inc ^= 6;
  }
  return true;
}

// adaptation of : http://stackoverflow.com/questions/4475996
//              (Howard Hinnant, Implementation 5)
// returns the smallest prime that is >= x
HashTable::size_type next_prime(HashTable::size_type x)
{
   switch (x)
   {
   case 0:
      return 1;
   case 1:
   case 2:
      return 2;
   case 3:
      return 3;
   case 4:
   case 5:
      return 5;
   }
   HashTable::size_type k = x / 6;
   HashTable::size_type i = x - 6 * k;
   HashTable::size_type inc = i < 2 ? 1 : 5;
   x = 6 * k + inc;
```

```
    for (i = (3 + inc) / 2; !is_prime(x); x += i)
        i ^= 6;
    return x;
}
```