Table Class

Specification and Implementation

Table Class

Specification

Table Class - Specification

Template parameters, typedefs, and member constants:

```
// template parameter
template <typename Key, typename Value>

// The table stores records, each of which consists of
// a Key and a Value.
//
// Value may be any built-in type, or any class that
// provides:
//  - instantiation via a default and copy constructors
//  - assignment operator (x = y)
```

Table Class - Specification

Template parameters, typedefs, and member constants:

```
// template parameter
template <typename Key, typename Value>


// Key may be any built-in type, or any class that
// provides:
//  - instantiation via a default and copy constructors
//  - assignment operator (x = y)
//  - operator ==(Key, Key)
//
// Further, a function called hash(Key) must exist and
// should return an integer >= 0
```

Table Class - Specification

Template parameters, typedefs, and member constants:

```
// the default maximum number of records in the table
static const std::size_t DEFAULT_CAPACITY = ________;
```

Table Class - Specification

Constructors:

```
    // creates an empty table
    Table(size_t max_records = DEFAULT_CAPACITY);


    // postcondition:
    //   the table has been initialized as an empty table,
    //   capable of storing up to max_records records
```

Table Class - Specification

Modification member functions:

```
// adds @value to the table, identified by @key
void insert(const Key& key, const Value& value);


// precondition:
//   hash(key) >= 0
//   size() < capacity
// postcondition:
//   If a record in the table with the given key already
//   exists, that record is replaced with value.
//   Otherwise, value has been added as a new record in
//   the table.
```

Table Class - Specification

Modification member functions:

```
// removes the record with the specified key
void remove(const Key& key);


// postcondition:
//   If a record was in the table with the specified
//   key, then that record has been removed. Otherwise,
//   the table is unchanged.
```

Table Class - Specification

Constant member functions:

```
// returns whether a record identified by @key exists
bool is_present(const Key& key) const;


// postcondition:
//   The return value is true if there is a record in
//   the table with the specified key. Otherwise, the
//   return value is false.
```

Table Class - Specification

Constant member functions:

```
// finds a record identified by @key, if it exists
bool find(const Key& key, Value& result) const;

// postcondition:
//   If a record is in the table with the specified key,
//   the return value is true and result is set to a
//   copy of the record with that key. Otherwise, the
//   return value is false and the value of result is
//   unchanged.
```

Table Class - Specification

Constant member functions:

```
// returns the number of records in the table
size_t size() const;


// postcondition:
//   The return value is the total number of records in
//   the table
```

Table Class - Specification

Value semantics:

```
// Table<Key, Value> objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

Table Class - Specification

Dynamic memory usage:

```
// If there is insufficient dynamic memory, then the
// following functions throw bad_alloc:
//   constructors
//   insert
//   operator =
```

Table Class Implementation

Table Class - Implementation

Starting the header file:

```
// Table.h header file

// specification documentation

#pragma once

#include <cstdlib>


namespace CS262 {
    template <typename Key, typename Value>
    class Table { };
}


#include "table.cpp"
```

Table Class - Implementation

Starting the implementation file:

```
// Table.cpp implementation file

// INVARIANT (coming soon)

#include <cassert>

#include <algorithm>

using namespace std;


namespace CS262 {

    // member implementations

}
```

Table Class - Implementation

Class skeleton, constants, and member variables:

```
template <typename Key, typename Value>
class Table {
    private: // Record class
        struct Record { Key key; Value value; };

    private: // data members
        Record** data;
        size_t num_records, capacity;
        static const int NEVER_USED      = 0;
        static const int PREVIOUSLY_USED = 1;
};
```

Table Class - Implementation

Document invariant in implementation file:

```
// Table.cpp implementation file
// This file is included in the header file and not
//   compiled separately.
// INVARIANT for Table<Key, Value> class:
// 1. The records in the table are stored in a
//    dynamically array called data, whose size is
//    stored in the member variable capacity
// 2. The number of records in the table is stored in
//    the member variable num_records
```

Table Class - Implementation

Document invariant in implementation file:

```
// 3. Individual records are stored with their key and
//    value together in a Record object's key and value
//    properties, respectively
// 4. The index at which a Record is stored depends on
//    the value of hash(record.key) % capacity; in the
//    event of a conflict, the next available index is
//    used instead (open-address hashing)
```

Table Class - Implementation

Document invariant in implementation file:

```
// 5. Each element in the data array is a pointer to a
//    Record object.
// 6. Never-occupied slots are indicated by the value
//    NEVER_USED; previously used slots are indicated by
//    the value PREVIOUSLY_USED; occupied slots contain
//    addresses of dynamically allocated Record objects
```

Point 6 addresses the use of these two constants (more on them later):

```
static const int NEVER_USED     = 0;
static const int PREVIOUSLY_USED = 1;
```

Table Class - Implementation

Some helper functions to make life easier (and more readable):

```
// returns the index of a key by "hashing" it
std::size_t index_hash(const Key& key) const {
    return (hash(key) % capacity);
}
```

This function:

- hashes the given key using the global hash function for the Key data type and converts it to an index in the array using the mod operator
- for example, to use a string as the key for our table, the user would have to provide the following function:

```
std::size_t hash(const string&); // hashes a string
```

Table Class - Implementation

Some helper functions to make life easier (and more readable):

```
// returns the index one after @index
std::size_t next_index(std::size_t index) const {
    return ((index + 1) % capacity);
}
```

This function:

- returns the next index after the given one
- it handles wrapping around to the beginning of the array when necessary

Table Class - Implementation

Some helper functions to make life easier (and more readable):

```
// returns whether @index has never been used before
bool never_used(std::size_t index) const {
    return data[index] == NEVER_USED;
}
```

This function:

- returns true if the array has never held a record at the given index, or false if it has

Table Class - Implementation

Some helper functions to make life easier (and more readable):

```
// returns whether or not @index is currently vacant
bool is_vacant(std::size_t index) const {
    return data[index] == NEVER_USED ||
           data[index] == PREVIOUSLY_USED;
}
```

This function:

- returns true if the element at data[index] is currently vacant
- this means it has either never been used or is previously used but is once again free

Table Class - Implementation

Some helper functions to make life easier (and more readable):

```
// gets index of the record with key @k, if it exists
bool find_index(const Key& k, std::size_t& i) const;
```

This function:

- if a record exists in the table with the specified key, returns true and i is set to the index of that record
- otherwise, it returns false, and the value of i is unchanged

Table Class - Implementation

Some helper functions to make life easier (and more readable):

```
    // gets index of the record with key @k, if it exists
    bool find_index(const Key& k, std::size_t& i) const {
        std::size_t count = 0;
        i = index_hash(k);
        while (!never_used(i) && data[i].key != key) {
            i = next_index(i);
            if (++count == capacity) break;
        }
        return (data[i].key == key);
}
```

Table Class - Implementation

The constructor:

```
// creates an empty table
Table(size_t max_records = DEFAULT_CAPACITY);
```

What it needs to do:

- initialize `capacity` to the value of the `max_records` argument
- set `max_records` to zero (initially an empty table)
- initialize `data` by allocating an array of `capacity` elements, all set to `NEVER_USED`

Table Class - Implementation

The constructor implementation:

```
template <typename Key, typename Value>
Table<Key, Value>::Table(size_t max_records) :
    num_records(0), capacity(max_records)
{
    // dynamically allocate an array of Record pointers
    data = new Record*[capacity];

    // set each element to NEVER_USED
    for (size_t i = 0; i < capacity; i++)
        data[i] = NEVER_USED;
}
```

Table Class - Implementation

The size method (inline implementation):

```
// returns the total number of records in the table
size_type size() const { return used; }
```

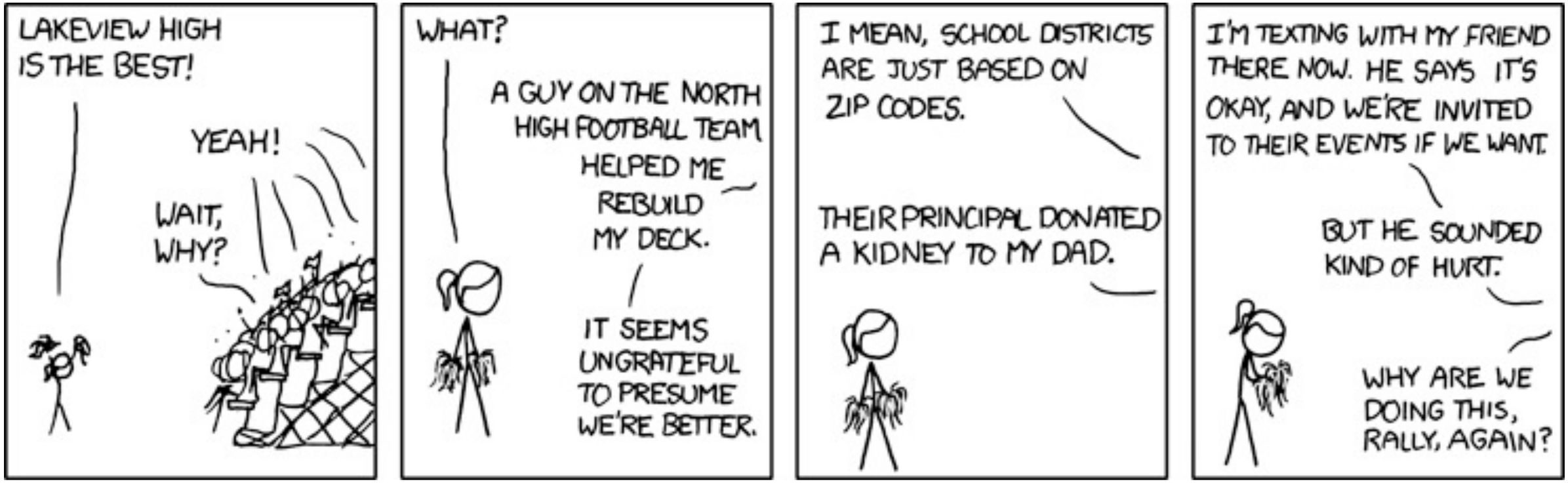

Table Class - Implementation

The insert method:

```
// adds @value to the table, identified by @key
void insert(const Key& key, const Value& value);
```

What it needs to do:

- make sure there's room to insert the value
- find the index where the value should go, or whether a record with the given key exists
- if no record in the table with the given key exists, add it to the table
- otherwise, overwrite the existing value
- increment num_records

Table Class - Implementation

The abbreviated insert implementation (missing template / class prefix):

```
void insert(const Key& key, const Value& value) {
    std::size_t i;

    if (!find_index(key, i)) {
        assert(size() < capacity);
        i = index_hash(key);
        while (!is_vacant(i)) i = next_index(i);
        num_records++;
    }
    data[i] = entry;
}
```

Table Class - Implementation

The remove method:

```
// removes the record with the specified key
void remove(const Key& key);
```

What it needs to do:

- find the index of the record with the given key, if it exists
- if it does, remove the record and set its index to PREVIOUSLY_USED
- decrement num_records
- if no such record, no action is necessary

Table Class - Implementation

The remove implementation:

```
template <typename Key, typename Value>
void Table<Key, Value>::remove(const Key& key);
    std::size_t i;

    if (find_index(key, i)) {
        delete data[i];
        data[i] = PREVIOUSLY_USED;
        num_records--;
    }
}
```

Table Class - Implementation

Constant member functions:

```
// returns whether a record identified by @key exists
bool is_present(const Key& key) const;


// finds a record identified by @key, if it exists
bool find(const Key& key, Value& result) const;
```

Using the helper methods shown earlier, these should be easy

- try implementing them if you want
- yeah, I know: you're not actually going to... >.>

Table Class - Implementation

The Table class as shown uses dynamic memory

- this means that we need to implement the big 3 for it
- I leave this as an exercise for your very capable skills =)

Cartoon!