Part I

Part 1 was fairly straightforward. Linear probing lets us make several assumptions about its behavior. For one, from any arbitrary array slot, we can probe to every other slot in exactly M iterations (where M is the map capacity). No infinite loops to worry about! The trickiest part of part 1 was ensuring that, once we remove an item from the array, we probe forward and remove/re-insert each item in the cluster. If I needed to implement a quick and dirty hashing implementation, I would likely choose open addressing with linear probing as my first choice unless there was a good reason that it did not work well for my use-case.

Part II

I really enjoyed part 2, simply because I love working with linked lists. At each slot, I hold on to a head pointer to the first item in the chain, and include a tail dummy node at the end, but I don't store a pointer to the tail. This makes it easy to update the chain. For instance, when I insert a new item, I traverse until I either find the key (in which case I simply update that node's value), or encounter the tail dummy node. If I hit the tail, that means the key was not yet stored in the list, and I transform the tail into a regular node holding the new item and append a new tail.

When removing, I simply traverse the list until I either hit the tail (item not found), or find an item. In that case, I replace the current item with it's next node using a copy assignment operator, and delete the next node's pointer.

Part III Bucket

Part 3 introduced new key types, which were all straightforward to implement except for the c-strings. I hate passing pointers around, especially if they come from the user and refer to character arrays that may go out of scope and get. What I did to solve this was wrap all keys and values in their own class, Key and Value respectively. These classes use another wrapper class, GenericContainer, to hold their raw value. I use template specialization to treat c-strings as special. When GenericContainer is initialized with a c-string, it makes a copy for itself. Because Keys and Values can be replaced during normal map operations, we don't want to lose the pointers. Therefore, whenever a raw value must persist (ie, when returning the value to the client), I create another copy of the raw value and pass the pointer to the client. The client is then responsible for deleting the c-string if he wishing to avoid memory leaks. This strategy allows me to abstract away the pointer logic, greatly reducing code complexity.

In addition, part III requested a new method, cluster_distribution. To implement this, I create an integer array of size M (where M is the map capacity()). Each index in that array refers to a cluster size, and the associated value is used as a counter to track the number of encountered cluster instances of that size. Once I take inventory of every cluster, I traverse that array and pass extant clusters to a priority queue and return that, which sorts the clusters by size in N lg N time. The client can then read each cluster inventory item off the priority queue in order.

Aside from that, the bucket aspect of part 3 was failry straightforward and used essentially the same architecture as part 2.

Part III Open Addressing

Part 3's open addressing section was the most difficult aspect of the entire project. It turns out that implementing an effective hashing method is very tricky, and I settled on fairly rudimentary functions for each of the supported keys. I experimented with functors - structs that expose the function call operators and rely upon function overloading for behavior specific to their types. This is how the c++ standard library seems to support plug-and-play coding, and I found it very effective.

I allow the client to specify three functors - a map capacity planner, a primary hash, and a secondary hash. The default map capacity planner takes the minimum capacity desired by the client, and returns the next-highest prime number to use as the true map capacity.

I was not able to get the remove() methods working for the quadratic probing or the double hashing instances. Quadratic probing tends to have a nasty habit of not reliably visiting every potential slot. Double hashing uses the key itself to pick the next slot, which leads to the following scenario:

- Keys A and B give equalivalent primary hashing values but different double hashing values
- Key A exists in slot 1
- Key B is inserted, skipping slot 1 and probing forward to slot 2
- Key A is removed, and afterwards we attempt to resolve the cluster and reposition Key B so that it can be visited. However, Key A's probing value is dependent on Key A, so it is impossible to know that Key B originally intended to take Key A's slot, so when subsequently searching for Key B we will visit an empty slot where A was and encounter an empty slot

I'm sure there is a way to get the remove() methods working with double hashing and quadratic probing, but I ran out of time, and honestly I got really tired of fighting with the subtle mathematical landmine that hashing functions turns out to be. I have a new-found respect for cryptography researchers who deal with that kind of thing all the time.

Part IV

Part 4 was really fun. I implemented a BST base class and then exposed that to RBST and AVL. RBST simply overrode the insertion functionality to randomly insert at the root. Implementing all the methods for RBST was an exercise in recursive thinking, especially the pretty-print function which I was able to implement. I got a little carried away with playing with the pretty-print function, and I wanted to stress-test it to see how far I could push it. Because I allocate the output buffer (an array of lines) and write to it in memory before dumping it to standard output, a simple array requires contiguous memory and did not support the hundreds of millions of nodes that I wanted to print. Therefore, I used CDAL from the last project to store large chunks of lines together, but in different parts of memory.

As mentioned in the testing strategies, the BST class includes a few methods to manually and recursively calculate the height and number of children at each subtree if the _DEBUG_preprocessor macro is set to true. This is disabled by default because it is an expensive operation, but the unit tests set it to true, which makes ensuring the tree maintains correct structure throughout its lifetime.

Part IV Bonus: AVL Tree

The AVL tree was a bit tricky because I had to keep track of the height at every operation. I solved this once I went through and determined every operation that could potentially change the height of the subtree. Since each operation that could potentially change the height of a subtree was a recursive function, the base case would only change the height by a maximum of one. Therefore, at the base case, I call an update_height method, which simply calculates the height of the current node as one plus the maximum height between the two child nodes. After that operation completes, the recursion bubbles up, and the parent subtree does the same thing.

An elegant consequence of this strategy is that the AVL tree is simply the BST, but with the insert() and remove() functions wrapped in code that balances the current subtree at every recursion level. The balance() method is the largest block of code in the class, which ends up being relatively short and easy to read through and see what's going on. In addition, I wrap a few other BST methods with code that, if _DEBUG_is set to true, will recursively verify that abs(balance factor) is less than or equal to one at every subtree. Like the code that does the same for height and number of children, this is an expensive operation and is disabled by default, but it is very useful for unit testing.

Part I: Hashmap with Open Addressing

Part 1 Testing Strategy

Paul Nickerson

I attempted to separate tests into two somewhat distinct categories: operations that are expected to fail (operation_failures.cpp), and operations that are expected to succeed (operation_successes.cpp).

operation_failures.cpp

Within the failure operations scenario, I start with an empty map and try to search() and remove() an item whose key does not exist in the map. Both calls should return false. I then fill the map to capacity, which is possible to do since linear probing allows us to reliably fill every slot, and attempt to insert() a key (should fail due to lack of space), remove() a key that doesn't exist in the map, and search() for a key that does not exist in the map.

operation_successes.cpp

The success-expecting operations is much more extensive. I start by filling the map halfway, clearing it, then filling it up halfway again. The map should then report the correct size. I check that several keys which are expected to exist in the map actually do exist (including the lowest possible key, the highest possible key, and one in the middle).

I check the print() function by routing it to an output string stream and count the number of hyphens in the output, which indicate empty slots. I ensure that these match the value of capacity() - size().

I then attempt to remove() several keys which are known to exist, and check that their associated values are what were expected. After these items are removed, I try to both search() and remove() them, which should all return false.

part1/part1.pdf

part1/checklist.txt,

```
Hashmap with Open Addressing written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part I: hashmaps with Open Addressing
=============================================================================
My MAP implementation uses the data structure described in the part I
instructions and conforms to the technique required for this map
variety: yes

My MAP implementation 100% correctly supports the following methods
as described in part I:

* insert: yes
* remove: yes
* search: yes
* clear: yes
* is_empty: yes
* capacity: yes
* size: yes
* load: yes
* print: yes

=============================================================================
FOR ALL PARTS
=============================================================================

My MAP implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My TEST compiles correctly using g++ v4.8.2 on the OpenBSD VM: yes

My TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this hashmaps with Open Addressing
and the associated tests.
Paul Nickerson, Dec 2 in COP3520 section 1087

-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
```

```
How to compile and run my unit tests on the OpenBSD VM
cd part1/source
./compile.sh
./run_tests > output.txt
```

common/common.h

common/common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#ifndef _DEBUG_
    //various internal integrity checks that can be expensive, we want to disable them in production
    #define _DEBUG_ false
#endif

#include <string.h>
#include <limits>
#include <stdexcept>
#include <ostream>
#include <cmath>

namespace cop3530 {
    inline double lg(size_t i) {
        return std::log(i) / std::log(2);
    }
    inline size_t rand_i(size_t max) {
        size_t bucket_size = RAND_MAX / max;
        size_t num_buckets = RAND_MAX / bucket_size;
        size_t big_rand;
        do {
               big_rand = rand();
        } while(big_rand >= num_buckets * bucket_size);
        return big_rand / bucket_size;
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        static constexpr size_t primes[] = { //from algorithms in c++, helps us to choose a prime-number map capacity
            251,
            509,
            1021,
            2039,
            4093,
            8191,
            16381,
            32749,
            65521,
            131071,
            262193,
            524287,
            1048573,
            2097143,
```

```
            4194301,
            8388593,
            16777213,
            33554393,
            67108859,
            134217689,
            268435399,
            536870909,
            1073741789,
            2147483647
        };
        struct ClusterInventory {
            size_t cluster_size;
            size_t num_instances;
            struct cluster_size_less_predicate {
                bool operator()(ClusterInventory const& cluster1, ClusterInventory
                     const& cluster2) {
                    return cluster1.cluster_size < cluster2.cluster_size;
                }
            };
        };

        inline size_t str_to_numeric(const char* str) {
            unsigned int base = 257; //prime number chosen near an 8-bit character
            size_t numeric = 0;
            for (; *str != 0; ++str)
                numeric = numeric * base + *str;
            return numeric;
        }
        namespace functors {

            struct map_capacity_planner {
                size_t operator()(size_t min_capacity) {
                    for (int i = 0; i != 24; ++i)
                        if (min_capacity < primes[i])
                            return primes[i];
                    throw std::domain_error("Provided min capacity too large.
                         Consider extending the list of prime numbers");
                }
            };
            struct compare {
                int operator()(const char* a, const char* b) const {
                    int cmp = strcmp(a, b);
                    return (cmp < 0 ? -1 :
                                    (cmp > 0 ? 1 : 0));
                }
                int operator()(double a, double b) const {
                    return (a < b ? -1 :
                                  (a > b ? 1 : 0));
                }
                int operator()(std::string const& a, std::string const& b) const {
                    return (a < b ? -1 :
```

```
                        (a > b ? 1 : 0));
    }
    int operator()(int a, int b) const {
        return (a < b ? -1 :
                        (a > b ? 1 : 0));
    }
};
namespace primary_hashes {
    struct hash_basic {
    //this is such a stupid hash method, but unlike my pathetic attempts
        at implementing
    //various other hashing methods, it works and is generalizable to
        all the required key
    //types. together with double hashing it should make for a passable
        hashing routine.
    public:
        size_t operator()(const char* key) const {
            return str_to_numeric(key);
        }
        size_t operator()(double key) const {
            return static_cast<size_t>(std::fmod(key, max_size_t));
        }
        size_t operator()(int key) const {
            return static_cast<size_t>(key);
        }
        size_t operator()(std::string const& key) const {
            const char* c_key = key.c_str();
            return operator()(c_key);
        }
    };
}
namespace secondary_hashes {
    struct linear_probe {
        bool changes_with_probe_attempt() const {
            return false;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return 1;
        }
    };
    struct quadratic_probe {
        bool changes_with_probe_attempt() const {
            return true;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return probe_attempt;
        }
    };
    struct hash_double {
    private:
```

```
145                 size_t hash_numeric(size_t numeric) const {
146                     size_t hash = numeric % 97; //simple modulus using a prime
                            number (from algorithms in c++)
147                     //the second hash may not be zero (will cause an infinite
                            loop).
148                     //also, hash must be relatively prime to map_capacity so that
                            every slot can be hit.
149                     //map capacity is a prime number based chosen from the table,
                            so any value less than
150                     //map capacity should work
151                     return hash;
152                 }
153             public:
154                 bool changes_with_probe_attempt() const {
155                     return false;
156                 }
157                 size_t operator()(const char* key, size_t) const {
158                     size_t numeric = str_to_numeric(key);
159                     return hash_numeric(numeric);
160                 }
161                 size_t operator()(double key, size_t) const {
162                     return hash_numeric(key);
163                 }
164                 size_t operator()(int key, size_t) const {
165                     return hash_numeric(key);
166                 }
167                 size_t operator()(std::string key, size_t) const {
168                     const char* c_key = key.c_str();
169                     return operator()(c_key, 0);
170                 }
171             };
172         }
173     }
174
175     template<typename T>
176     class GenericContainer {
177         /*
178             for the types we need to support other than const char* (ie int,
                    double, and std::string),
179             we can pass these around willy-nilly. for const char*, handled
                    below, we will obtain our
180             own copy of the character array by wrapping it in a std::string
181         */
182     private:
183         T raw;
184         functors::compare compare;
185     public:
186         GenericContainer(const T& val): raw(val) {}
187         GenericContainer() = default;
188         GenericContainer& operator=(GenericContainer const& rhs) = delete;
189         T operator()() const {
190             return raw;
```

```
191             }
192             T copy() const {
193                 return raw;
194             }
195             void reset(const T& val) {
196                 raw = val;
197             }
198             int compare_to(GenericContainer const& other) const {
199                 return compare(raw, other.raw);
200             }
201         };
202         template<>
203         class GenericContainer<const char*> {
204             /*
205                 class template specialization for character arrays, stores a local
                        copy of the character array
206             */
207         private:
208             char* raw = nullptr;
209             functors::compare compare;
210         public:
211             GenericContainer(const char* val) {
212                 reset(val);
213             }
214             GenericContainer() = default;
215             const char* operator()() const {
216                 return raw;
217             }
218             const char* copy() const {
219                 if (raw == nullptr) return nullptr;
220                 size_t len = strlen(raw);
221                 char* new_str = new char[len + 1];
222                 strncpy(new_str, raw, len);
223                 new_str[len] = 0;
224                 return new_str;
225             }
226             void reset(const char* val) {
227                 if (raw) {
228                     delete raw;
229                     raw = nullptr;
230                 }
231                 if (val != nullptr) {
232                     size_t len = strlen(val);
233                     raw = new char[len + 1];
234                     strncpy(raw, val, len);
235                     raw[len] = 0;
236                 }
237             }
238             int compare_to(GenericContainer const& other) const {
239                 return compare(raw, other.raw);
240             }
241         };
```

```
242
243         template<typename key_type,
244                  typename primary_hash =
                         hash_utils::functors::primary_hashes::hash_basic,
245                  typename secondary_hash =
                         hash_utils::functors::secondary_hashes::hash_double>
246         class Key {
247         private:
248             GenericContainer<key_type> raw_key;
249             primary_hash hasher1;
250             secondary_hash hasher2;
251             size_t hash1_val;
252             size_t hash2_val;
253         public:
254             Key& operator=(Key const& rhs) {
255                 if (&rhs == this)
256                     return *this;
257                 reset(rhs.raw_key());
258             }
259             bool operator==(Key const& rhs) const {
260                 return raw_key.compare_to(rhs.raw_key) == 0;
261             }
262             bool operator<(Key const& rhs) const {
263                 return raw_key.compare_to(rhs.raw_key) == -1;
264             }
265             bool operator>(Key const& rhs) const {
266                 return raw_key.compare_to(rhs.raw_key) == 1;
267             }
268             bool operator!=(Key const& rhs) const {
269                 return ! operator==(rhs);
270             }
271             size_t hash(size_t map_capacity, size_t probe_attempt) const {
272                 size_t local_hash2_val;
273                 if (probe_attempt != 0 && hasher2.changes_with_probe_attempt())
274                 {
275                     //if the hashing function value is dependent on the probe attempt
276                     //(eg quadratic probing), then we need to retrieve the new value*/
277                     local_hash2_val = hasher2(raw_key(), probe_attempt);
278                 } else {
279                     //otherwise we can just use the value we have stored
280                     local_hash2_val = hash2_val;
281                 }
282                 return (hash1_val + probe_attempt * local_hash2_val) % map_capacity;
283             }
284             key_type raw() const {
285                 return raw_key();
286             }
287             key_type raw_copy() const {
288                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
289                 return raw_key.copy();
290             }
```

```
291             template<typename T>
292             void reset(T key) {
293                 raw_key.reset(key);
294                 size_t base_probe_attempt = 0;
295                 hash1_val = hasher1(key);
296                 hash2_val = hasher2(key, base_probe_attempt);
297             }
298             void reset(const char* key) {
299                 raw_key.reset(key);
300                 if (key != nullptr) {
301                     size_t base_probe_attempt = 0;
302                     hash1_val = hasher1(key);
303                     hash2_val = hasher2(key, base_probe_attempt);
304                 }
305             }
306             explicit Key(key_type const& key): raw_key(key) {
307                 reset(key);
308             }
309             Key() = default;
310         };
311         template <typename value_type>
312         class Value {
313         private:
314             functors::compare compare;
315             GenericContainer<value_type> raw_value;
316         public:
317             Value& operator=(Value const& rhs) {
318                 if (&rhs == this)
319                     return *this;
320                 reset(rhs.raw_value());
321             }
322             bool operator==(Value const& rhs) const {
323                 return compare(raw_value(), rhs.raw_value());
324             }
325             bool operator==(value_type const& rhs) const {
326                 return compare(raw_value(), rhs) == 0;
327             }
328             value_type raw() const {
329                 return raw_value();
330             }
331             value_type raw_copy() const {
332                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
333                 return raw_value.copy();
334             }
335             void reset(value_type value) {
336                 raw_value.reset(value);
337             }
338             explicit Value(value_type const& value): raw_value(value) {}
339             Value() = default;
340         };
341     }
```

```
342 }
343
344 #endif
```

common/priority_queue.h

common/priority_queue.h

```
#ifndef _PRIORITY_QUEUE_H_
#define _PRIORITY_QUEUE_H_

#include "SDAL.h"
#include "common.h"

namespace cop3530 {
    //this class allows efficient sorting clusters by size for the
        cluster_distribution functions
    template<typename T,
            typename PriorityCompare =
                cop3530::hash_utils::ClusterInventory::cluster_size_less_predicate>
    class priority_queue {
    private:
        PriorityCompare first_arg_higher_priority;
        //SDAL has all the benefits of std::vector (ie fast random access and
            automatic resizing)
        //while having the added benefit of being legal to use in cop3530
        SDAL<T> tree;
        size_t num_items = 0;
        void fix_up(size_t index) {
            while (index > 1
                  && first_arg_higher_priority(tree[index], tree[index / 2]))
            {
                std::swap(tree[index / 2], tree[index]);
                index /= 2;
            }
        }
        void fix_down() {
            size_t parent_index = 1;
            while (2 * parent_index <= num_items) {
                size_t left_index = 2 * parent_index;
                size_t right_index = left_index + 1;
                size_t higher_priority_index = left_index;
                if (right_index <= num_items
                    && first_arg_higher_priority(tree[right_index], tree[left_index]))
                {
                    higher_priority_index = right_index;
                }
                if ( ! first_arg_higher_priority(tree[higher_priority_index],
                     tree[parent_index]))
                    //no more items to elevate
                    break;
                std::swap(tree[parent_index], tree[higher_priority_index]);
                parent_index = higher_priority_index;
            }
        }
```

```
44      public:
45          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
46              T empty_item;
47              tree.push_back(empty_item);
48          }
49          priority_queue(priority_queue const& src) {
50              tree = src.tree;
51              num_items = src.num_items;
52          }
53          T get_next_item() {
54              std::swap(tree[1], tree[num_items]);
55              T ret = tree[num_items--];
56              fix_down();
57              return ret;
58          }
59          void add_to_queue(T const& item) {
60              tree.push_back(item);
61              num_items++;
62              fix_up(num_items);
63          }
64          size_t size() {
65              return num_items;
66          }
67          bool empty() {
68              return num_items == 0;
69          }
70      };
71  }
72
73  #endif // _PRIORITY_QUEUE_H_
```

part1/source/open_addressing_map.h

part1/source/open_addressing_map.h

```
#ifndef _OPEN_ADDRESSING_MAP_H_
#define _OPEN_ADDRESSING_MAP_H_

#include <iostream>
#include "../../common/common.h"
#include <stdexcept>

namespace cop3530 {
    class HashMapOpenAddressing {
    private:
        typedef int key_type;
        typedef char value_type;
        typedef hash_utils::ClusterInventory ClusterInventory;
        struct Slot {
            key_type key;
            value_type value;
            bool is_occupied = false;
        };
        Slot* slots;
        size_t curr_capacity = 0;
        size_t num_occupied_slots = 0;
        size_t probe(size_t i) {
            return i;
        }
        size_t hash(key_type const& key) {
            size_t M = capacity();
            hash_utils::functors::primary_hashes::hash_basic hasher;
            size_t big_hash_number = hasher(key);
            size_t hash_val = big_hash_number % M;
            return hash_val;
        }
        /*
            searches the map for an item matching key. returns the number of probe
                attempts needed
            to reach either the item or an empty slot
        */
        int search_internal(key_type const& key) {
            size_t M = capacity();
            size_t hash_val = hash(key);
            size_t probe_index;
            for (probe_index = 0; probe_index != M; ++probe_index) {
                size_t slot_index = (hash_val + probe(probe_index)) % M;
                if (slots[slot_index].is_occupied) {
                    if (slots[slot_index].key == key) {
                        //found the key
                        break;
                    }
```

```
47                } else
48                    //found unoccupied slot
49                    break;
50            }
51            return 1 + probe_index; //start with a single probe when probe_index==0
52        }
53        //all backing array manipulations should go through the following two
              methods
54        void insert_at_index(key_type const& key, value_type const& value, size_t
              index) {
55            Slot& s = slots[index];
56            s.key = key;
57            s.value = value;
58            if ( ! s.is_occupied) {
59                s.is_occupied = true;
60                ++num_occupied_slots;
61            }
62        }
63        value_type remove_at_index(size_t index) {
64            Slot& s = slots[index];
65            if (s.is_occupied) {
66                s.is_occupied = false;
67                --num_occupied_slots;
68            }
69            return s.value;
70        }
71    public:
72        HashMapOpenAddressing(size_t const min_capacity)
73        {
74            if (min_capacity == 0) {
75                throw std::domain_error("min_capacity must be at least 1");
76            }
77            cop3530::hash_utils::functors::map_capacity_planner capacity_planner;
78            curr_capacity = capacity_planner(min_capacity); //make capacity prime
79            slots = new Slot[curr_capacity];
80        }
81        ~HashMapOpenAddressing() {
82            delete slots;
83        }
84        /*
85            if there is space available, adds the specified key/value-pair to the
                  hash map and returns true; otherwise
86            returns false. If an item already exists in the map with the same key,
                  replace its value.
87        */
88        bool insert(key_type const& key, value_type const& value) {
89            size_t M = capacity();
90            if (M == size())
91                return false;
92            size_t probes_required = search_internal(key);
93            size_t index = (hash(key) + probe(probes_required - 1)) % M;
94            insert_at_index(key, value, index);
```

```
95              return true;
96          }
97          /*
98              if there is an item matching key, removes the key/value-pair from the
                    map, stores it's value in value,
99              and returns true; otherwise returns false.
100         */
101         bool remove(key_type const& key, value_type& value) {
102             size_t M = capacity();
103             size_t probes_required = search_internal(key);
104             size_t index = (hash(key) + probe(probes_required - 1)) % M;
105             if ( ! (slots[index].is_occupied && slots[index].key == key))
106                 //key not found
107                 return false;
108             value = remove_at_index(index);
109             size_t start_index = index;
110             //remove and reinsert items until find unoccupied slot
111             for (int i = 1; ; ++i) {
112                 index = (start_index + probe(i)) % M;
113                 Slot const& s = slots[index];
114                 if (s.is_occupied) {
115                     remove_at_index(index);
116                     insert(s.key, s.value);
117                 } else {
118                     break;
119                 }
120             }
121             return true;
122         }
123         /*
124             if there is an item matching key, stores it's value in value,
125             and returns true (the item remains in the map); otherwise returns false.
126         */
127         bool search(key_type const& key, value_type& value) {
128             size_t M = capacity();
129             size_t probes_required = search_internal(key);
130             size_t index = (hash(key) + probe(probes_required - 1)) % M;
131             if ( ! (slots[index].is_occupied && slots[index].key == key))
132                 //key not found
133                 return false;
134             value = slots[index].value;
135             return true;
136         }
137         /*
138             removes all items from the map.
139         */
140         void clear() {
141             size_t cap = capacity();
142             for (size_t i = 0; i != cap; ++i)
143                 slots[i].is_occupied = false;
144             num_occupied_slots = 0;
145         }
```

```
146         /*
147             returns true IFF the map contains no elements.
148         */
149         bool is_empty() {
150             return size() == 0;
151         }
152         /*
153             returns the number of slots in the map.
154         */
155         size_t capacity() {
156             return curr_capacity;
157         }
158         /*
159             returns the number of items actually stored in the map.
160         */
161         size_t size() {
162             return num_occupied_slots;
163         }
164         /*
165             returns the map's load factor (size = load * capacity).
166         */
167         double load() {
168             return static_cast<double>(size()) / capacity();
169         }
170         /*
171             inserts into the ostream, the backing array's contents in sequential
                order.
172             Empty slots shall be denoted by a hyphen, non-empty slots by that item's
173             key. [This function will be used for debugging/monitoring].
174         */
175         std::ostream& print(std::ostream& out) {
176             size_t cap = capacity();
177             out << '[';
178             for (size_t i = 0; i != cap; ++i) {
179                 if (slots[i].is_occupied) {
180                     out << slots[i].key;
181                 } else {
182                     out << "-";
183                 }
184                 if (i + 1 < cap)
185                     out << '|';
186             }
187             out << ']';
188             return out;
189         }
190 
191     };
192 }
193 
194 #endif
```

Part II: Hashmap with Buckets

Part 2 Testing Strategy

Paul Nickerson

I attempted to separate tests into two somewhat distinct categories: operations that are expected to fail (operation_failures.cpp), and operations that are expected to succeed (operation_successes.cpp).

operation_failures.cpp

Within the failure operations scenario, I start with an empty map and try to search() and remove() an item whose key does not exist in the map. Both calls should return false. I then fill the map with a bunch of items (it is impossible to run out of space because collisions are resolved via an arbitrarily-growable liked list). From this newly-filled map, I attempt to remove() a key that doesn't exist in the map, and search() for a key that does not exist in the map, both of which should return false.

operation_successes.cpp

The success-expecting operations is much more extensive. I start by filling the map with a bunch of items, clearing it, then filling it up again. The map should then report the correct size. I check that several keys which are expected to exist in the map actually do exist (including the lowest possible key, the highest possible key, and one in the middle).

I check the print() function by routing it to an output string stream and count the number of hyphens in the output, which indicate empty slots. Since load factor = occupied buckets / capacity, we can get the number of unoccupied buckets as capacity * (1 - load). This should equal the number of hyphens in the print() output.

I then attempt to remove() several keys which are known to exist, and check that their associated values are what were expected. After these items are removed, I try to both search() and remove() them, which should all return false.

part2/part2.pdf

part2/checklist.txt

```
Hashmaps with Buckets written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part II: Hashmaps with Buckets
=============================================================================
My MAP implementation uses the data structure described in the part II
instructions and conforms to the technique required for this map
variety: yes

My MAP implementation 100% correctly supports the following methods
as described in part I:

* insert: yes
* remove: yes
* search: yes
* clear: yes
* is_empty: yes
* capacity: yes
* size: yes
* load: yes
* print: yes

=============================================================================
FOR ALL PARTS
=============================================================================

My MAP implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My TEST compiles correctly using g++ v4.8.2 on the OpenBSD VM: yes

My TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this Hashmaps with Buckets
and the associated tests.
Paul Nickerson, Dec 2 in COP3520 section 1087

-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
```

```
How to compile and run my unit tests on the OpenBSD VM
cd part2/source
./compile.sh
./run_tests > output.txt
```

common/common.h

common/common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#ifndef _DEBUG_
    //various internal integrity checks that can be expensive, we want to disable them in production
    #define _DEBUG_ false
#endif

#include <string.h>
#include <limits>
#include <stdexcept>
#include <ostream>
#include <cmath>

namespace cop3530 {
    inline double lg(size_t i) {
        return std::log(i) / std::log(2);
    }
    inline size_t rand_i(size_t max) {
        size_t bucket_size = RAND_MAX / max;
        size_t num_buckets = RAND_MAX / bucket_size;
        size_t big_rand;
        do {
               big_rand = rand();
        } while(big_rand >= num_buckets * bucket_size);
        return big_rand / bucket_size;
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        static constexpr size_t primes[] = { //from algorithms in c++, helps us to choose a prime-number map capacity
            251,
            509,
            1021,
            2039,
            4093,
            8191,
            16381,
            32749,
            65521,
            131071,
            262193,
            524287,
            1048573,
            2097143,
```

```
            4194301,
            8388593,
            16777213,
            33554393,
            67108859,
            134217689,
            268435399,
            536870909,
            1073741789,
            2147483647
        };
        struct ClusterInventory {
            size_t cluster_size;
            size_t num_instances;
            struct cluster_size_less_predicate {
                bool operator()(ClusterInventory const& cluster1, ClusterInventory
                     const& cluster2) {
                    return cluster1.cluster_size < cluster2.cluster_size;
                }
            };
        };

        inline size_t str_to_numeric(const char* str) {
            unsigned int base = 257; //prime number chosen near an 8-bit character
            size_t numeric = 0;
            for (; *str != 0; ++str)
                numeric = numeric * base + *str;
            return numeric;
        }
        namespace functors {

            struct map_capacity_planner {
                size_t operator()(size_t min_capacity) {
                    for (int i = 0; i != 24; ++i)
                        if (min_capacity < primes[i])
                            return primes[i];
                    throw std::domain_error("Provided min capacity too large.
                         Consider extending the list of prime numbers");
                }
            };
            struct compare {
                int operator()(const char* a, const char* b) const {
                    int cmp = strcmp(a, b);
                    return (cmp < 0 ? -1 :
                                    (cmp > 0 ? 1 : 0));
                }
                int operator()(double a, double b) const {
                    return (a < b ? -1 :
                                  (a > b ? 1 : 0));
                }
                int operator()(std::string const& a, std::string const& b) const {
                    return (a < b ? -1 :
```

```
                        (a > b ? 1 : 0));
    }
    int operator()(int a, int b) const {
        return (a < b ? -1 :
                        (a > b ? 1 : 0));
    }
};
namespace primary_hashes {
    struct hash_basic {
    //this is such a stupid hash method, but unlike my pathetic attempts
        at implementing
    //various other hashing methods, it works and is generalizable to
        all the required key
    //types. together with double hashing it should make for a passable
        hashing routine.
    public:
        size_t operator()(const char* key) const {
            return str_to_numeric(key);
        }
        size_t operator()(double key) const {
            return static_cast<size_t>(std::fmod(key, max_size_t));
        }
        size_t operator()(int key) const {
            return static_cast<size_t>(key);
        }
        size_t operator()(std::string const& key) const {
            const char* c_key = key.c_str();
            return operator()(c_key);
        }
    };
}
namespace secondary_hashes {
    struct linear_probe {
        bool changes_with_probe_attempt() const {
            return false;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return 1;
        }
    };
    struct quadratic_probe {
        bool changes_with_probe_attempt() const {
            return true;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return probe_attempt;
        }
    };
    struct hash_double {
    private:
```

```
145                 size_t hash_numeric(size_t numeric) const {
146                     size_t hash = numeric % 97; //simple modulus using a prime
                            number (from algorithms in c++)
147                     //the second hash may not be zero (will cause an infinite
                            loop).
148                     //also, hash must be relatively prime to map_capacity so that
                            every slot can be hit.
149                     //map capacity is a prime number based chosen from the table,
                            so any value less than
150                     //map capacity should work
151                     return hash;
152                 }
153             public:
154                 bool changes_with_probe_attempt() const {
155                     return false;
156                 }
157                 size_t operator()(const char* key, size_t) const {
158                     size_t numeric = str_to_numeric(key);
159                     return hash_numeric(numeric);
160                 }
161                 size_t operator()(double key, size_t) const {
162                     return hash_numeric(key);
163                 }
164                 size_t operator()(int key, size_t) const {
165                     return hash_numeric(key);
166                 }
167                 size_t operator()(std::string key, size_t) const {
168                     const char* c_key = key.c_str();
169                     return operator()(c_key, 0);
170                 }
171             };
172         }
173     }
174
175     template<typename T>
176     class GenericContainer {
177         /*
178             for the types we need to support other than const char* (ie int,
                    double, and std::string),
179             we can pass these around willy-nilly. for const char*, handled
                    below, we will obtain our
180             own copy of the character array by wrapping it in a std::string
181         */
182     private:
183         T raw;
184         functors::compare compare;
185     public:
186         GenericContainer(const T& val): raw(val) {}
187         GenericContainer() = default;
188         GenericContainer& operator=(GenericContainer const& rhs) = delete;
189         T operator()() const {
190             return raw;
```

```
191             }
192             T copy() const {
193                 return raw;
194             }
195             void reset(const T& val) {
196                 raw = val;
197             }
198             int compare_to(GenericContainer const& other) const {
199                 return compare(raw, other.raw);
200             }
201         };
202         template<>
203         class GenericContainer<const char*> {
204             /*
205                 class template specialization for character arrays, stores a local
                        copy of the character array
206             */
207         private:
208             char* raw = nullptr;
209             functors::compare compare;
210         public:
211             GenericContainer(const char* val) {
212                 reset(val);
213             }
214             GenericContainer() = default;
215             const char* operator()() const {
216                 return raw;
217             }
218             const char* copy() const {
219                 if (raw == nullptr) return nullptr;
220                 size_t len = strlen(raw);
221                 char* new_str = new char[len + 1];
222                 strncpy(new_str, raw, len);
223                 new_str[len] = 0;
224                 return new_str;
225             }
226             void reset(const char* val) {
227                 if (raw) {
228                     delete raw;
229                     raw = nullptr;
230                 }
231                 if (val != nullptr) {
232                     size_t len = strlen(val);
233                     raw = new char[len + 1];
234                     strncpy(raw, val, len);
235                     raw[len] = 0;
236                 }
237             }
238             int compare_to(GenericContainer const& other) const {
239                 return compare(raw, other.raw);
240             }
241         };
```

```
242
243         template<typename key_type,
244                  typename primary_hash =
                             hash_utils::functors::primary_hashes::hash_basic,
245                  typename secondary_hash =
                             hash_utils::functors::secondary_hashes::hash_double>
246         class Key {
247         private:
248             GenericContainer<key_type> raw_key;
249             primary_hash hasher1;
250             secondary_hash hasher2;
251             size_t hash1_val;
252             size_t hash2_val;
253         public:
254             Key& operator=(Key const& rhs) {
255                 if (&rhs == this)
256                     return *this;
257                 reset(rhs.raw_key());
258             }
259             bool operator==(Key const& rhs) const {
260                 return raw_key.compare_to(rhs.raw_key) == 0;
261             }
262             bool operator<(Key const& rhs) const {
263                 return raw_key.compare_to(rhs.raw_key) == -1;
264             }
265             bool operator>(Key const& rhs) const {
266                 return raw_key.compare_to(rhs.raw_key) == 1;
267             }
268             bool operator!=(Key const& rhs) const {
269                 return ! operator==(rhs);
270             }
271             size_t hash(size_t map_capacity, size_t probe_attempt) const {
272                 size_t local_hash2_val;
273                 if (probe_attempt != 0 && hasher2.changes_with_probe_attempt())
274                 {
275                     //if the hashing function value is dependent on the probe attempt
276                     //(eg quadratic probing), then we need to retrieve the new value*/
277                     local_hash2_val = hasher2(raw_key(), probe_attempt);
278                 } else {
279                     //otherwise we can just use the value we have stored
280                     local_hash2_val = hash2_val;
281                 }
282                 return (hash1_val + probe_attempt * local_hash2_val) % map_capacity;
283             }
284             key_type raw() const {
285                 return raw_key();
286             }
287             key_type raw_copy() const {
288                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
289                 return raw_key.copy();
290             }
```

```
291             template<typename T>
292             void reset(T key) {
293                 raw_key.reset(key);
294                 size_t base_probe_attempt = 0;
295                 hash1_val = hasher1(key);
296                 hash2_val = hasher2(key, base_probe_attempt);
297             }
298             void reset(const char* key) {
299                 raw_key.reset(key);
300                 if (key != nullptr) {
301                     size_t base_probe_attempt = 0;
302                     hash1_val = hasher1(key);
303                     hash2_val = hasher2(key, base_probe_attempt);
304                 }
305             }
306             explicit Key(key_type const& key): raw_key(key) {
307                 reset(key);
308             }
309             Key() = default;
310         };
311         template <typename value_type>
312         class Value {
313         private:
314             functors::compare compare;
315             GenericContainer<value_type> raw_value;
316         public:
317             Value& operator=(Value const& rhs) {
318                 if (&rhs == this)
319                     return *this;
320                 reset(rhs.raw_value());
321             }
322             bool operator==(Value const& rhs) const {
323                 return compare(raw_value(), rhs.raw_value());
324             }
325             bool operator==(value_type const& rhs) const {
326                 return compare(raw_value(), rhs) == 0;
327             }
328             value_type raw() const {
329                 return raw_value();
330             }
331             value_type raw_copy() const {
332                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
333                 return raw_value.copy();
334             }
335             void reset(value_type value) {
336                 raw_value.reset(value);
337             }
338             explicit Value(value_type const& value): raw_value(value) {}
339             Value() = default;
340         };
341     }
```

```
342 }
343
344 #endif
```

common/priority_queue.h

common/priority_queue.h

```
#ifndef _PRIORITY_QUEUE_H_
#define _PRIORITY_QUEUE_H_

#include "SDAL.h"
#include "common.h"

namespace cop3530 {
    //this class allows efficient sorting clusters by size for the
        cluster_distribution functions
    template<typename T,
            typename PriorityCompare =
                cop3530::hash_utils::ClusterInventory::cluster_size_less_predicate>
    class priority_queue {
    private:
        PriorityCompare first_arg_higher_priority;
        //SDAL has all the benefits of std::vector (ie fast random access and
            automatic resizing)
        //while having the added benefit of being legal to use in cop3530
        SDAL<T> tree;
        size_t num_items = 0;
        void fix_up(size_t index) {
            while (index > 1
                  && first_arg_higher_priority(tree[index], tree[index / 2]))
            {
                std::swap(tree[index / 2], tree[index]);
                index /= 2;
            }
        }
        void fix_down() {
            size_t parent_index = 1;
            while (2 * parent_index <= num_items) {
                size_t left_index = 2 * parent_index;
                size_t right_index = left_index + 1;
                size_t higher_priority_index = left_index;
                if (right_index <= num_items
                    && first_arg_higher_priority(tree[right_index], tree[left_index]))
                {
                    higher_priority_index = right_index;
                }
                if ( ! first_arg_higher_priority(tree[higher_priority_index],
                     tree[parent_index]))
                    //no more items to elevate
                    break;
                std::swap(tree[parent_index], tree[higher_priority_index]);
                parent_index = higher_priority_index;
            }
        }
```

```
44      public:
45          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
46              T empty_item;
47              tree.push_back(empty_item);
48          }
49          priority_queue(priority_queue const& src) {
50              tree = src.tree;
51              num_items = src.num_items;
52          }
53          T get_next_item() {
54              std::swap(tree[1], tree[num_items]);
55              T ret = tree[num_items--];
56              fix_down();
57              return ret;
58          }
59          void add_to_queue(T const& item) {
60              tree.push_back(item);
61              num_items++;
62              fix_up(num_items);
63          }
64          size_t size() {
65              return num_items;
66          }
67          bool empty() {
68              return num_items == 0;
69          }
70      };
71  }
72
73  #endif // _PRIORITY_QUEUE_H_
```

part2/source/buckets_map.h

part2/source/buckets_map.h

```
#ifndef _BUCKETS_MAP_H_
#define _BUCKETS_MAP_H_

#include <iostream>
#include <stdexcept>
#include "../../common/common.h"

namespace cop3530 {
    class HashMapBuckets {
    private:
        typedef int key_type;
        typedef char value_type;
        typedef hash_utils::ClusterInventory ClusterInventory;
        struct Item {
            key_type key;
            value_type value;
            Item* next;
            bool is_dummy;
            Item(Item* next): next(next), is_dummy(true) {}
        };
        struct Bucket {
            Item* head; //use a head pointer to the first node, and include a dummy
                node at the end (but dont store its pointer)
            Bucket() {
                Item* tail = new Item(nullptr);
                head = tail;
            }
            ~Bucket() {
                while ( ! head->is_dummy) {
                    Item* to_delete = head;
                    head = head->next;
                    delete to_delete;
                }
                delete head; //tail
            }
        };
        typedef Item* link;
        Bucket* buckets;
        size_t num_buckets = 0;
        size_t num_items = 0;
        size_t hash(key_type const& key) {
            size_t M = capacity();
            hash_utils::functors::primary_hashes::hash_basic hasher;
            return hasher(key) % M;
        }
        /*
            searches the bucket corresponding to the specified key's hash for that
```

```
47              key. if found, stores a reference to that item and returns P, the number
                    of
48              probe attempts needed to get to the item (ie the number of chain links
                    needed
49              to be traversed). otherwise return -1 * P and stores the pointer to the
                    tail dummy node in
50              item_ptr.
51          */
52          int search_internal(key_type const& key, link& item_ptr) {
53              int probe_attempts = 1;
54              size_t hash_val = hash(key);
55              Bucket& bucket = buckets[hash_val];
56              item_ptr = bucket.head;
57              while ( ! item_ptr->is_dummy) {
58                  if (item_ptr->key == key) {
59                      //found the key
60                      return probe_attempts;
61                  }
62                  item_ptr = item_ptr->next;
63                  ++probe_attempts;
64              }
65              //key not found
66              return probe_attempts * -1;
67          }
68          void init() {
69              buckets = new Bucket[num_buckets];
70              num_items = 0;
71          }
72      public:
73          HashMapBuckets(size_t const min_buckets)
74          {
75              if (min_buckets == 0) {
76                  throw std::domain_error("min_buckets must be at least 1");
77              }
78              cop3530::hash_utils::functors::map_capacity_planner capacity_planner;
79              num_buckets = capacity_planner(min_buckets); //make capacity prime
80              init();
81          }
82          ~HashMapBuckets() {
83              delete[] buckets;
84          }
85          /*
86              if there is space available, adds the specified key/value-pair to the
                    hash map and returns true; otherwise
87              returns false. If an item already exists in the map with the same key,
                    replace its value.
88              note: this will never return false because we add to a linked list to
                    resolve collisions
89          */
90          bool insert(key_type const& key, value_type const& value) {
91              Item* item;
92              int probes_required = search_internal(key, item);
```

```
93              if (probes_required > 0)
94                  //found item
95                  item->value = value;
96              else {
97                  //currently holding tail (item not found). transform it into a valid
                        item then add a new tail
98                  item->is_dummy = false;
99                  item->key = key;
100                 item->value = value;
101                 item->next = new Item(nullptr);
102                 ++num_items;
103             }
104             return true;
105         }
106         /*
107             if there is an item matching key, removes the key/value-pair from the
                    map, stores it's
108             value in value, and returns true; otherwise returns false.
109         */
110         bool remove(key_type const& key, value_type& value) {
111             Item* item;
112             int probes_required = search_internal(key, item);
113             if (probes_required > 0) {
114                 //found item
115                 value = item->value;
116                 //swap the current item for the next one
117                 Item* to_delete = item->next;
118                 *item = *to_delete;
119                 delete to_delete;
120                 --num_items;
121                 return true;
122             }
123             return false;
124         }
125         /*
126             if there is an item matching key, stores it's value in value, and
                    returns true (the
127             item remains in the map); otherwise returns false.
128         */
129         bool search(key_type const& key, value_type& value) {
130             Item* item;
131             int probes_required = search_internal(key, item);
132             if (probes_required > 0) {
133                 //found item
134                 value = item->value;
135                 return true;
136             }
137             return false;
138         }
139         /*
140             removes all items from the map.
141         */
```

```
142         void clear() {
143             delete[] buckets;
144             init();
145         }
146         /*
147             returns true IFF the map contains no elements.
148         */
149         bool is_empty() {
150             return size() == 0;
151         }
152         /*
153             returns the number of slots in the map.
154         */
155         size_t capacity() {
156             return num_buckets;
157         }
158         /*
159             returns the number of items actually stored in the map.
160         */
161         size_t size() {
162             return num_items;
163         }
164         /*
165             returns the map's load factor (occupied buckets = load * capacity).
166         */
167         double load() {
168             size_t occupied_buckets = 0;
169             if (size() > 0) {
170                 size_t M = capacity();
171                 for (size_t i = 0; i != M; ++i) {
172                     Bucket const& bucket = buckets[i];
173                     if ( ! bucket.head->is_dummy)
174                         //bucket has at least one item
175                         occupied_buckets++;
176                 }
177             }
178             return static_cast<double>(occupied_buckets) / capacity();
179         }
180         /*
181             inserts into the ostream, the backing array's contents in sequential
                order.
182             Empty slots shall be denoted by a hyphen, non-empty slots by that item's
183             key. [This function will be used for debugging/monitoring].
184         */
185         std::ostream& print(std::ostream& out) {
186             size_t cap = capacity();
187             bool print_separator = false;
188             out << '[';
189             for (size_t i = 0; i != cap; ++i) {
190                 Bucket const& bucket = buckets[i];
191                 if (bucket.head->is_dummy) {
192                     if (print_separator)
```

```
193                         out << "|";
194                     else
195                         print_separator = true;
196                     out << "-";
197                 } else {
198                     for (Item* item = bucket.head; item->is_dummy != true; item =
                         item->next) {
199                         if (print_separator)
200                             out << "|";
201                         else
202                             print_separator = true;
203                         out << item->key;
204                     }
205                 }
206             }
207             out << ']';
208             return out;
209         }
210     };
211 }
212
213 #endif
```

Part III: Parameterizable Hashmap with Open Addressing

Part 3 Open Addressing Testing Strategy

Paul Nickerson

In addition to separating tests by operations that are expected to fail and those that are expected to succeed, I also included a scenario for testing the cluster_distribution() function.

operation_failures_(probe type/double hash).cpp

I include three failure operations scenarios, one for each of the secondary hashing methods to be supported (linear probing, quadratic probing, and double hashing). Within each scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string).

Linear Probing

To the linear probing instance, I start with an empty map and try to search() and remove() an item whose key does not exist in the map. Both calls should return a value less than zero (indicating key not found). I then fill the map to capacity, and attempt to insert an item where there is no space for it, which is expected to return a value less than zero. From this newly-filled map, I attempt to remove() a key that doesn't exist in the map, and search() for a key that does not exist in the map, both of which should return a value less than zero.

Quadratic Probing and Double Hashing

I was not able to get the remove() methods working for the quadratic probing or the double hashing instances. Quadratic probing tends to have a nasty habit of not reliably visiting every potential slot. Double hashing uses the key itself to pick the next slot, which leads to the following scenario:

- Keys A and B give equalivalent primary hashing values but different double hashing values
- Key A exists in slot 1
- Key B is inserted, skipping slot 1 and probing forward to slot 2
- Key A is removed, and afterwards we attempt to resolve the cluster and reposition Key B so that it can be visited. However, Key A's probing value is dependent on Key A, so it is impossible to know that Key B originally intended to take Key A's slot, so

when subsequently searching for Key B we will visit an empty slot where A was and encounter an empty slot

Because of these issues, I did not test the remove() functionality of the quadratic probing and double hashing instances, opting instead to simply fill the map to capacity and then search for a key that should not exist in the map. Note that quadratic hashing, as previously mentioned, has a tendency to sometimes ignore some slots, and the probability of this occuring increases drasticly as the load factor approaches 1. I early-abandon when this becomes evident to avoid an infinite loop, but, as a result, it was not possible to include a test to try and insert into a completely filled map like I could do in linear probing, since one key may induce an infinite loop while another key successfully finds one of the last remaining slots.

operation_successes_(probe type/double hash).cpp

Within the three success operations scenarios (one for each of the hashing methods), I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start by filling the map with a bunch of items, clearing it, then filling it up again. The map should then report the correct size. I check that several keys which are expected to exist in the map actually do exist (including the lowest possible key, the highest possible key, and one in the middle).

I check the print() function by routing it to an output string stream and count the number of hyphens in the output, which indicate empty slots. The number of hyphens in the print() output should equal capacity() - size(), ie the number of unoccupied slots.

As previously mentioned, I was unable to successfully implement remove() functionality in quadratic probing and double hashing, but within linear probing I attempt to remove() several keys which are known to exist, and check that their associated values are what were expected. After these items are removed, I try to both search() and remove() them, which should all return false.

cluster_distribution_(probe type/double hash).cpp

Since cluster_distribution() returns a priority queue of clusters, and each cluster has a minimum size of one, all the clusters taken together should encompass every occupied slot. Therefore, I fill the map with a bunch of items, then clear it and fill it again to try and destablize the map. From there, I take the summation, over every cluster, of the cluster's size times the number of clusters having that size. The result of that summation should equal the output from the map's size() method. I do this four times for each of the hashing method instances, once for each of the key types supported (a total of 12 times).

part3/open_addressing/part3open.pdf

part3/open_addressing/checklist.txt

```
hashmaps with Open Addressing written by Nickerson, Paul
COP 3530, 2014F 1087
=========================================================================
Part III: hashmaps with Open Addressing
=========================================================================
My MAP implementation uses the data structure described in the part II
instructions and conforms to the technique required for this map
variety: yes

My MAP implementation 100% correctly supports all three probing
techniques: (no - everything works except removing items with double hashing and quadratic probi

My MAP implementation 100% correctly supports the following key types:
* signed int: yes
* double: yes
* c-string: yes
* std::string: yes

My MAP implementation 100% correctly supports the ANY value type: yes

My MAP implementation 100% correctly supports the following methods
as described in part I:

* clear: yes
* is_empty: yes
* capacity: yes
* size: yes
* load: yes
* print: yes

My MAP implementation 100% correctly supports the following revised
and new methods as described in part III:

* insert: yes
* remove: (linear probing: yes, quadratic probing: no, double hashing: no)
* search: yes
* cluster_distribution(): yes
* remove_random(): (linear probing: yes, quadratic probing: no, double hashing: no)

=========================================================================
FOR ALL PARTS
=========================================================================
```

```
My MAP implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My TEST compiles correctly using g++ v4.8.2 on the OpenBSD VM: yes

My TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this hashmaps with Open Addressing
and the associated tests.
Paul Nickerson, Dec 2 in COP3520 section 1087

-------------------------------------------------------------------------
-------------------------------------------------------------------------

How to compile and run my unit tests on the OpenBSD VM
cd part3/open_addressing/source
./compile.sh
./run_tests > output.txt
```

common/common.h

common/common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#ifndef _DEBUG_
    //various internal integrity checks that can be expensive, we want to disable them in production
    #define _DEBUG_ false
#endif

#include <string.h>
#include <limits>
#include <stdexcept>
#include <ostream>
#include <cmath>

namespace cop3530 {
    inline double lg(size_t i) {
        return std::log(i) / std::log(2);
    }
    inline size_t rand_i(size_t max) {
        size_t bucket_size = RAND_MAX / max;
        size_t num_buckets = RAND_MAX / bucket_size;
        size_t big_rand;
        do {
               big_rand = rand();
        } while(big_rand >= num_buckets * bucket_size);
        return big_rand / bucket_size;
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        static constexpr size_t primes[] = { //from algorithms in c++, helps us to choose a prime-number map capacity
            251,
            509,
            1021,
            2039,
            4093,
            8191,
            16381,
            32749,
            65521,
            131071,
            262193,
            524287,
            1048573,
            2097143,
```

```
            4194301,
            8388593,
            16777213,
            33554393,
            67108859,
            134217689,
            268435399,
            536870909,
            1073741789,
            2147483647
        };
        struct ClusterInventory {
            size_t cluster_size;
            size_t num_instances;
            struct cluster_size_less_predicate {
                bool operator()(ClusterInventory const& cluster1, ClusterInventory
                     const& cluster2) {
                    return cluster1.cluster_size < cluster2.cluster_size;
                }
            };
        };

        inline size_t str_to_numeric(const char* str) {
            unsigned int base = 257; //prime number chosen near an 8-bit character
            size_t numeric = 0;
            for (; *str != 0; ++str)
                numeric = numeric * base + *str;
            return numeric;
        }
        namespace functors {

            struct map_capacity_planner {
                size_t operator()(size_t min_capacity) {
                    for (int i = 0; i != 24; ++i)
                        if (min_capacity < primes[i])
                            return primes[i];
                    throw std::domain_error("Provided min capacity too large.
                         Consider extending the list of prime numbers");
                }
            };
            struct compare {
                int operator()(const char* a, const char* b) const {
                    int cmp = strcmp(a, b);
                    return (cmp < 0 ? -1 :
                                    (cmp > 0 ? 1 : 0));
                }
                int operator()(double a, double b) const {
                    return (a < b ? -1 :
                                  (a > b ? 1 : 0));
                }
                int operator()(std::string const& a, std::string const& b) const {
                    return (a < b ? -1 :
```

```
                        (a > b ? 1 : 0));
    }
    int operator()(int a, int b) const {
        return (a < b ? -1 :
                        (a > b ? 1 : 0));
    }
};
namespace primary_hashes {
    struct hash_basic {
    //this is such a stupid hash method, but unlike my pathetic attempts
        at implementing
    //various other hashing methods, it works and is generalizable to
        all the required key
    //types. together with double hashing it should make for a passable
        hashing routine.
    public:
        size_t operator()(const char* key) const {
            return str_to_numeric(key);
        }
        size_t operator()(double key) const {
            return static_cast<size_t>(std::fmod(key, max_size_t));
        }
        size_t operator()(int key) const {
            return static_cast<size_t>(key);
        }
        size_t operator()(std::string const& key) const {
            const char* c_key = key.c_str();
            return operator()(c_key);
        }
    };
}
namespace secondary_hashes {
    struct linear_probe {
        bool changes_with_probe_attempt() const {
            return false;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return 1;
        }
    };
    struct quadratic_probe {
        bool changes_with_probe_attempt() const {
            return true;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return probe_attempt;
        }
    };
    struct hash_double {
    private:
```

```
145                 size_t hash_numeric(size_t numeric) const {
146                     size_t hash = numeric % 97; //simple modulus using a prime
                            number (from algorithms in c++)
147                     //the second hash may not be zero (will cause an infinite
                            loop).
148                     //also, hash must be relatively prime to map_capacity so that
                            every slot can be hit.
149                     //map capacity is a prime number based chosen from the table,
                            so any value less than
150                     //map capacity should work
151                     return hash;
152                 }
153             public:
154                 bool changes_with_probe_attempt() const {
155                     return false;
156                 }
157                 size_t operator()(const char* key, size_t) const {
158                     size_t numeric = str_to_numeric(key);
159                     return hash_numeric(numeric);
160                 }
161                 size_t operator()(double key, size_t) const {
162                     return hash_numeric(key);
163                 }
164                 size_t operator()(int key, size_t) const {
165                     return hash_numeric(key);
166                 }
167                 size_t operator()(std::string key, size_t) const {
168                     const char* c_key = key.c_str();
169                     return operator()(c_key, 0);
170                 }
171             };
172         }
173     }
174
175     template<typename T>
176     class GenericContainer {
177         /*
178             for the types we need to support other than const char* (ie int,
                    double, and std::string),
179             we can pass these around willy-nilly. for const char*, handled
                    below, we will obtain our
180             own copy of the character array by wrapping it in a std::string
181         */
182     private:
183         T raw;
184         functors::compare compare;
185     public:
186         GenericContainer(const T& val): raw(val) {}
187         GenericContainer() = default;
188         GenericContainer& operator=(GenericContainer const& rhs) = delete;
189         T operator()() const {
190             return raw;
```

```
191             }
192             T copy() const {
193                 return raw;
194             }
195             void reset(const T& val) {
196                 raw = val;
197             }
198             int compare_to(GenericContainer const& other) const {
199                 return compare(raw, other.raw);
200             }
201         };
202         template<>
203         class GenericContainer<const char*> {
204             /*
205                 class template specialization for character arrays, stores a local
                        copy of the character array
206             */
207         private:
208             char* raw = nullptr;
209             functors::compare compare;
210         public:
211             GenericContainer(const char* val) {
212                 reset(val);
213             }
214             GenericContainer() = default;
215             const char* operator()() const {
216                 return raw;
217             }
218             const char* copy() const {
219                 if (raw == nullptr) return nullptr;
220                 size_t len = strlen(raw);
221                 char* new_str = new char[len + 1];
222                 strncpy(new_str, raw, len);
223                 new_str[len] = 0;
224                 return new_str;
225             }
226             void reset(const char* val) {
227                 if (raw) {
228                     delete raw;
229                     raw = nullptr;
230                 }
231                 if (val != nullptr) {
232                     size_t len = strlen(val);
233                     raw = new char[len + 1];
234                     strncpy(raw, val, len);
235                     raw[len] = 0;
236                 }
237             }
238             int compare_to(GenericContainer const& other) const {
239                 return compare(raw, other.raw);
240             }
241         };
```

```
242
243         template<typename key_type,
244                  typename primary_hash =
                             hash_utils::functors::primary_hashes::hash_basic,
245                  typename secondary_hash =
                             hash_utils::functors::secondary_hashes::hash_double>
246         class Key {
247         private:
248             GenericContainer<key_type> raw_key;
249             primary_hash hasher1;
250             secondary_hash hasher2;
251             size_t hash1_val;
252             size_t hash2_val;
253         public:
254             Key& operator=(Key const& rhs) {
255                 if (&rhs == this)
256                     return *this;
257                 reset(rhs.raw_key());
258             }
259             bool operator==(Key const& rhs) const {
260                 return raw_key.compare_to(rhs.raw_key) == 0;
261             }
262             bool operator<(Key const& rhs) const {
263                 return raw_key.compare_to(rhs.raw_key) == -1;
264             }
265             bool operator>(Key const& rhs) const {
266                 return raw_key.compare_to(rhs.raw_key) == 1;
267             }
268             bool operator!=(Key const& rhs) const {
269                 return ! operator==(rhs);
270             }
271             size_t hash(size_t map_capacity, size_t probe_attempt) const {
272                 size_t local_hash2_val;
273                 if (probe_attempt != 0 && hasher2.changes_with_probe_attempt())
274                 {
275                     //if the hashing function value is dependent on the probe attempt
276                     //(eg quadratic probing), then we need to retrieve the new value*/
277                     local_hash2_val = hasher2(raw_key(), probe_attempt);
278                 } else {
279                     //otherwise we can just use the value we have stored
280                     local_hash2_val = hash2_val;
281                 }
282                 return (hash1_val + probe_attempt * local_hash2_val) % map_capacity;
283             }
284             key_type raw() const {
285                 return raw_key();
286             }
287             key_type raw_copy() const {
288                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
289                 return raw_key.copy();
290             }
```

```
291             template<typename T>
292             void reset(T key) {
293                 raw_key.reset(key);
294                 size_t base_probe_attempt = 0;
295                 hash1_val = hasher1(key);
296                 hash2_val = hasher2(key, base_probe_attempt);
297             }
298             void reset(const char* key) {
299                 raw_key.reset(key);
300                 if (key != nullptr) {
301                     size_t base_probe_attempt = 0;
302                     hash1_val = hasher1(key);
303                     hash2_val = hasher2(key, base_probe_attempt);
304                 }
305             }
306             explicit Key(key_type const& key): raw_key(key) {
307                 reset(key);
308             }
309             Key() = default;
310         };
311         template <typename value_type>
312         class Value {
313         private:
314             functors::compare compare;
315             GenericContainer<value_type> raw_value;
316         public:
317             Value& operator=(Value const& rhs) {
318                 if (&rhs == this)
319                     return *this;
320                 reset(rhs.raw_value());
321             }
322             bool operator==(Value const& rhs) const {
323                 return compare(raw_value(), rhs.raw_value());
324             }
325             bool operator==(value_type const& rhs) const {
326                 return compare(raw_value(), rhs) == 0;
327             }
328             value_type raw() const {
329                 return raw_value();
330             }
331             value_type raw_copy() const {
332                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
333                 return raw_value.copy();
334             }
335             void reset(value_type value) {
336                 raw_value.reset(value);
337             }
338             explicit Value(value_type const& value): raw_value(value) {}
339             Value() = default;
340         };
341     }
```

```
342 }
343
344 #endif
```

common/priority_queue.h

common/priority_queue.h

```
#ifndef _PRIORITY_QUEUE_H_
#define _PRIORITY_QUEUE_H_

#include "SDAL.h"
#include "common.h"

namespace cop3530 {
    //this class allows efficient sorting clusters by size for the
        cluster_distribution functions
    template<typename T,
            typename PriorityCompare =
                cop3530::hash_utils::ClusterInventory::cluster_size_less_predicate>
    class priority_queue {
    private:
        PriorityCompare first_arg_higher_priority;
        //SDAL has all the benefits of std::vector (ie fast random access and
            automatic resizing)
        //while having the added benefit of being legal to use in cop3530
        SDAL<T> tree;
        size_t num_items = 0;
        void fix_up(size_t index) {
            while (index > 1
                  && first_arg_higher_priority(tree[index], tree[index / 2]))
            {
                std::swap(tree[index / 2], tree[index]);
                index /= 2;
            }
        }
        void fix_down() {
            size_t parent_index = 1;
            while (2 * parent_index <= num_items) {
                size_t left_index = 2 * parent_index;
                size_t right_index = left_index + 1;
                size_t higher_priority_index = left_index;
                if (right_index <= num_items
                    && first_arg_higher_priority(tree[right_index], tree[left_index]))
                {
                    higher_priority_index = right_index;
                }
                if ( ! first_arg_higher_priority(tree[higher_priority_index],
                     tree[parent_index]))
                    //no more items to elevate
                    break;
                std::swap(tree[parent_index], tree[higher_priority_index]);
                parent_index = higher_priority_index;
            }
        }
```

```
44      public:
45          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
46              T empty_item;
47              tree.push_back(empty_item);
48          }
49          priority_queue(priority_queue const& src) {
50              tree = src.tree;
51              num_items = src.num_items;
52          }
53          T get_next_item() {
54              std::swap(tree[1], tree[num_items]);
55              T ret = tree[num_items--];
56              fix_down();
57              return ret;
58          }
59          void add_to_queue(T const& item) {
60              tree.push_back(item);
61              num_items++;
62              fix_up(num_items);
63          }
64          size_t size() {
65              return num_items;
66          }
67          bool empty() {
68              return num_items == 0;
69          }
70      };
71  }
72
73  #endif // _PRIORITY_QUEUE_H_
```

part3/open_addressing/source/open_addressing_generic_map.h

part3/open_addressing/source/open_addressing_generic_map.h

```
#ifndef _HASHMAPOPENADDRESSINGGENERIC_H_
#define _HASHMAPOPENADDRESSINGGENERIC_H_

#include <iostream>
#include <string>
#include "../../../common/common.h"
#include "../../../common/priority_queue.h"

namespace cop3530 {
    template<typename key_type,
             typename value_type,
             typename capacity_plan_functor =
                 hash_utils::functors::map_capacity_planner,
             typename primary_hash =
                 hash_utils::functors::primary_hashes::hash_basic,
             typename secondary_hash =
                 hash_utils::functors::secondary_hashes::hash_double>
    class HashMapOpenAddressingGeneric {
    private:
        typedef hash_utils::ClusterInventory ClusterInventory;
        typedef hash_utils::Key<key_type, primary_hash, secondary_hash> Key;
        typedef hash_utils::Value<value_type> Value;
        struct Item {
            Key key;
            Value value;
        };
        struct Slot {
            Item item;
            bool is_occupied = false;
        };
        Slot* slots;
        capacity_plan_functor choose_capacity;
        size_t curr_capacity = 0;
        size_t num_occupied_slots = 0;
        /*
            searches the map for an item matching key. returns the number of probe
                attempts needed
            to reach either the item or an empty slot
        */
        int search_internal(Key const& key) {
            size_t M = capacity();
            size_t probe_index;
            for (probe_index = 0; probe_index != M; ++probe_index) {
                size_t slot_index = key.hash(M, probe_index);
                if (slots[slot_index].is_occupied) {
                    if (slots[slot_index].item.key == key) {
                        //found the key
```

```
                    break;
                }
            } else
                //found unoccupied slot
                break;
        }
        return 1 + probe_index; //start with a single probe
    }

    //all backing array manipulations should go through the following two methods
    void insert_at_index(Key const& key, Value const& value, size_t index) {
        Slot& s = slots[index];
        s.item.key.reset(key.raw_copy());
        s.item.value.reset(value.raw_copy());
        if ( ! s.is_occupied) {
            s.is_occupied = true;
            ++num_occupied_slots;
        }
    }
    Value const& remove_at_index(size_t index) {
        Slot& s = slots[index];
        if (s.is_occupied) {
            s.is_occupied = false;
            --num_occupied_slots;
        }
        return s.item.value;
    }
public:
    HashMapOpenAddressingGeneric(size_t const min_capacity)
    {
        if (min_capacity == 0) {
            throw std::domain_error("min_capacity must be at least 1");
        }
        curr_capacity = choose_capacity(min_capacity);
        slots = new Slot[curr_capacity];
    }
    ~HashMapOpenAddressingGeneric() {
        delete[] slots;
    }

    /*
        if there is space available, adds the specified key/value-pair to the hash map and returns the
        number of probes required, P; otherwise returns -1 * P. If an item already exists in the map
        with the same key, replace its value.
    */
    int insert(key_type const& key, value_type const& value) {
        size_t M = capacity();
        if (M == size())
            return -1 * size();
```

```
93              Key k(key);
94              Value v(value);
95              int probes_required = search_internal(k);
96              size_t index = k.hash(M, probes_required - 1);
97              if (slots[index].is_occupied == true && slots[index].item.key != k)
98                  //map is full and we're going to hit an infinite loop if we keep
                        going
99                  return probes_required * -1;
100             insert_at_index(k, v, index);
101             return probes_required;
102         }
103
104         /*
105             if there is an item matching key, removes the key/value-pair from the
                    map, stores it's value in
106             value, and returns the number of probes required, P; otherwise returns
                    -1 * P.
107         */
108         int remove(key_type const& key, value_type& value) {
109             size_t M = capacity();
110             Key k(key);
111             int probes_required = search_internal(k);
112             size_t index = k.hash(M, probes_required - 1);
113             if (slots[index].is_occupied == false || slots[index].item.key != k)
114                 //key not found
115                 return -1 * probes_required;
116             Value v = remove_at_index(index);
117             value = v.raw_copy();
118             //remove and reinsert items until find unoccupied slot (guaranteed to
                    happen since we just removed an item)
119             for (int i = 1; ; ++i) {
120                 index = k.hash(M, i);
121                 Slot const& s = slots[index];
122                 if (s.is_occupied) {
123                     remove_at_index(index);
124                     insert(s.item.key.raw_copy(), s.item.value.raw_copy());
125                 } else {
126                     break;
127                 }
128             }
129             return probes_required;
130         }
131
132         /*
133             if there is an item matching key, stores it's value in value, and
                    returns the
134             number of probes required, P; otherwise returns -1 * P. Regardless, the
                    item
135             remains in the map.
136         */
137         int search(key_type const& key, value_type& value) {
138             size_t M = capacity();
```

```
139             Key k(key);
140             size_t probes_required = search_internal(k);
141             size_t index = k.hash(M, probes_required - 1);
142             if (slots[index].is_occupied == false || slots[index].item.key != k)
143                 //key not found
144                 return -1 * probes_required;
145             value = slots[index].item.value.raw_copy();
146             return probes_required;
147         }
148 
149         /*
150             removes all items from the map.
151         */
152         void clear() {
153             size_t cap = capacity();
154             for (size_t i = 0; i != cap; ++i)
155                 slots[i].is_occupied = false;
156             num_occupied_slots = 0;
157         }
158         /*
159             returns true IFF the map contains no elements.
160         */
161         bool is_empty() const {
162             return size() == 0;
163         }
164         /*
165             returns the number of slots in the map.
166         */
167         size_t capacity() const {
168             return curr_capacity;
169         }
170         /*
171             returns the number of items actually stored in the map.
172         */
173         size_t size() const {
174             return num_occupied_slots;
175         }
176         /*
177             returns the map's load factor (size = load * capacity).
178         */
179         double load() const {
180             return static_cast<double>(size()) / capacity();
181         }
182         /*
183             inserts into the ostream, the backing array's contents in sequential
                order.
184             Empty slots shall be denoted by a hyphen, non-empty slots by that item's
185             key. [This function will be used for debugging/monitoring].
186         */
187         std::ostream& print(std::ostream& out) const {
188             size_t cap = capacity();
189             out << '[';
```

```
190             for (size_t i = 0; i != cap; ++i) {
191                 if (slots[i].is_occupied) {
192                     out << slots[i].item.key.raw();
193                 } else {
194                     out << "-";
195                 }
196                 if (i + 1 < cap)
197                     out << '|';
198             }
199             out << ']';
200             return out;
201         }
202 
203         priority_queue<ClusterInventory> cluster_distribution() {
204             //use an array to count cluster instances, then feed those to a priority
                    queue and return it.
205             priority_queue<ClusterInventory> cluster_pq;
206             if (size() == 0) return cluster_pq;
207             size_t M = capacity();
208             size_t cluster_counter[M + 1];
209             for (size_t i = 0; i <= M; ++i)
210                 cluster_counter[i] = 0;
211             if (size() == M) {
212                 //handle the special case when the map is full
213                 cluster_counter[size()]++;
214             } else {
215                 //have at least one unoccupied slot
216                 bool first_cluster_skipped = false;
217                 size_t curr_cluster_size = 0;
218                 //treat the backing array as a circular buffer and make a maximum of
                        two passes to
219                 //capture everything, including the wraparound cluster if it exists
220                 for (size_t i = 1; i != M * 2; ++i) {
221                     Slot const& curr_slot = slots[i % M], prev_slot = slots[(i - 1) %
                            M];
222                     if (curr_slot.is_occupied && prev_slot.is_occupied) {
223                         //still in a cluster
224                         ++curr_cluster_size;
225                     } else if (curr_slot.is_occupied && prev_slot.is_occupied ==
                            false) {
226                         //found a new cluster
227                         curr_cluster_size = 1;
228                     } else if ( ! curr_slot.is_occupied && prev_slot.is_occupied) {
229                         //found the end of a cluster
230                         if (first_cluster_skipped) {
231                             cluster_counter[curr_cluster_size]++;
232                             if (i >= M) {
233                                 //reached the end of the first cluster in the second
                                        pass, so no all clusters have been handled
234                                 break;
235                             }
236                         } else {
```

```
                        first_cluster_skipped = true;
                    }
                }
            }
        }
        for (size_t i = 1; i <= M; ++i)
            if (cluster_counter[i] > 0) {
                ClusterInventory cluster{i, cluster_counter[i]};
                cluster_pq.add_to_queue(cluster);
            }
        return cluster_pq;
    }

    /*
        generate a random number, R, (1,size), and starting with slot zero in
            the backing array,
        find the R-th occupied slot; remove the item from that slot (adjusting
            subsequent items as
        necessary), and return its key.
    */
    key_type remove_random() {
        if (size() == 0) throw std::logic_error("Cant remove from an empty map");
        size_t num_slots = capacity();
        size_t ith_node_to_delete = 1 + rand_i(size());
        for (size_t i = 0; i != num_slots; ++i) {
            Slot const& slot = slots[i];
            if (slot.is_occupied && --ith_node_to_delete == 0) {
                key_type key = slot.item.key.raw_copy();
                value_type val_dummy;
                remove(key, val_dummy);
                return key;
            }
        }
        throw std::logic_error("Unexpected end of remove_random function");
    }
};
}

#endif
```

Part III: Parameterizable Hashmap with Buckets

Part 3 Buckets Testing Strategy

Paul Nickerson

In addition to separating tests by operations that are expected to fail and those that are expected to succeed, I also included a scenario for testing the cluster_distribution() function.

operation_failures.cpp

Within the failure operations scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start with an empty map and try to search() and remove() an item whose key does not exist in the map. Both calls should return a value less than zero (indicating key not found). I then fill the map with a bunch of items (it is impossible to run out of space because collisions are resolved via an arbitrarily-growable liked list). From this newly-filled map, I attempt to remove() a key that doesn't exist in the map, and search() for a key that does not exist in the map, both of which should return a value less than zero.

operation_successes.cpp

Within the success operations scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start by filling the map with a bunch of items, clearing it, then filling it up again. The map should then report the correct size. I check that several keys which are expected to exist in the map actually do exist (including the lowest possible key, the highest possible key, and one in the middle).

I check the print() function by routing it to an output string stream and count the number of hyphens in the output, which indicate empty slots. Since load factor = occupied buckets / capacity, we can get the number of unoccupied buckets as capacity * (1 - load). This should equal the number of hyphens in the print() output.

I then attempt to remove() several keys which are known to exist, and check that their associated values are what were expected. After these items are removed, I try to both search() and remove() them, which should all return false.

Part 3 added another function - remove_random() - which I test in a similar way to the preceeding remove() check. I remove a random key, then try to search() for it and remove() it. Both checks should fail and return a value less than zero.

cluster_distribution.cpp

Since cluster_distribution() returns a priority queue of clusters, and each cluster has a minimum size of one, all the clusters taken together should encompass every occupied slot. Therefore, I fill the map with a bunch of items, then clear it and fill it again to try and destablize the map. From there, I take the summation, over every cluster, of the cluster's size times the number of clusters having that size. The result of that summation should equal the output from the map's size() method. I do this four times, once for each of the four key types to be supported.

part3/bucket/part3bucket.pdf

part3/bucket/checklist.txt

```
Hashmaps with Buckets written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part III: Hashmaps with Buckets
=============================================================================
My MAP implementation uses the data structure described in the part II
instructions and conforms to the technique required for this map
variety: yes

My MAP implementation 100% correctly supports the following key types:
* signed int: yes
* double: yes
* c-string: yes
* std::string: yes

My MAP implementation 100% correctly supports the ANY value type: yes

My MAP implementation 100% correctly supports the following methods
as described in part I:

* clear: yes
* is_empty: yes
* capacity: yes
* size: yes
* load: yes
* print: yes

My MAP implementation 100% correctly supports the following revised
and new methods as described in part III:

* insert: yes
* remove: yes
* search: yes
* cluster_distribution(): yes
* remove_random(): yes
=============================================================================
FOR ALL PARTS
=============================================================================

My MAP implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My TEST compiles correctly using g++ v4.8.2 on the OpenBSD VM: yes
```

```
My TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this Hashmaps with Buckets
and the associated tests.
Paul Nickerson, Dec 2 in COP3520 section 1087

------------------------------------------------------------------------
------------------------------------------------------------------------

How to compile and run my unit tests on the OpenBSD VM
cd part3/bucket/source
./compile.sh
./run_tests > output.txt
```

common/common.h

common/common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#ifndef _DEBUG_
    //various internal integrity checks that can be expensive, we want to disable them in production
    #define _DEBUG_ false
#endif

#include <string.h>
#include <limits>
#include <stdexcept>
#include <ostream>
#include <cmath>

namespace cop3530 {
    inline double lg(size_t i) {
        return std::log(i) / std::log(2);
    }
    inline size_t rand_i(size_t max) {
        size_t bucket_size = RAND_MAX / max;
        size_t num_buckets = RAND_MAX / bucket_size;
        size_t big_rand;
        do {
               big_rand = rand();
        } while(big_rand >= num_buckets * bucket_size);
        return big_rand / bucket_size;
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        static constexpr size_t primes[] = { //from algorithms in c++, helps us to choose a prime-number map capacity
            251,
            509,
            1021,
            2039,
            4093,
            8191,
            16381,
            32749,
            65521,
            131071,
            262193,
            524287,
            1048573,
            2097143,
```

```
            4194301,
            8388593,
            16777213,
            33554393,
            67108859,
            134217689,
            268435399,
            536870909,
            1073741789,
            2147483647
        };
        struct ClusterInventory {
            size_t cluster_size;
            size_t num_instances;
            struct cluster_size_less_predicate {
                bool operator()(ClusterInventory const& cluster1, ClusterInventory
                     const& cluster2) {
                    return cluster1.cluster_size < cluster2.cluster_size;
                }
            };
        };

        inline size_t str_to_numeric(const char* str) {
            unsigned int base = 257; //prime number chosen near an 8-bit character
            size_t numeric = 0;
            for (; *str != 0; ++str)
                numeric = numeric * base + *str;
            return numeric;
        }
        namespace functors {

            struct map_capacity_planner {
                size_t operator()(size_t min_capacity) {
                    for (int i = 0; i != 24; ++i)
                        if (min_capacity < primes[i])
                            return primes[i];
                    throw std::domain_error("Provided min capacity too large.
                         Consider extending the list of prime numbers");
                }
            };
            struct compare {
                int operator()(const char* a, const char* b) const {
                    int cmp = strcmp(a, b);
                    return (cmp < 0 ? -1 :
                                    (cmp > 0 ? 1 : 0));
                }
                int operator()(double a, double b) const {
                    return (a < b ? -1 :
                                  (a > b ? 1 : 0));
                }
                int operator()(std::string const& a, std::string const& b) const {
                    return (a < b ? -1 :
```

```
                        (a > b ? 1 : 0));
    }
    int operator()(int a, int b) const {
        return (a < b ? -1 :
                        (a > b ? 1 : 0));
    }
};
namespace primary_hashes {
    struct hash_basic {
    //this is such a stupid hash method, but unlike my pathetic attempts
        at implementing
    //various other hashing methods, it works and is generalizable to
        all the required key
    //types. together with double hashing it should make for a passable
        hashing routine.
    public:
        size_t operator()(const char* key) const {
            return str_to_numeric(key);
        }
        size_t operator()(double key) const {
            return static_cast<size_t>(std::fmod(key, max_size_t));
        }
        size_t operator()(int key) const {
            return static_cast<size_t>(key);
        }
        size_t operator()(std::string const& key) const {
            const char* c_key = key.c_str();
            return operator()(c_key);
        }
    };
}
namespace secondary_hashes {
    struct linear_probe {
        bool changes_with_probe_attempt() const {
            return false;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return 1;
        }
    };
    struct quadratic_probe {
        bool changes_with_probe_attempt() const {
            return true;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return probe_attempt;
        }
    };
    struct hash_double {
    private:
```

```
145                 size_t hash_numeric(size_t numeric) const {
146                     size_t hash = numeric % 97; //simple modulus using a prime
                            number (from algorithms in c++)
147                     //the second hash may not be zero (will cause an infinite
                            loop).
148                     //also, hash must be relatively prime to map_capacity so that
                            every slot can be hit.
149                     //map capacity is a prime number based chosen from the table,
                            so any value less than
150                     //map capacity should work
151                     return hash;
152                 }
153             public:
154                 bool changes_with_probe_attempt() const {
155                     return false;
156                 }
157                 size_t operator()(const char* key, size_t) const {
158                     size_t numeric = str_to_numeric(key);
159                     return hash_numeric(numeric);
160                 }
161                 size_t operator()(double key, size_t) const {
162                     return hash_numeric(key);
163                 }
164                 size_t operator()(int key, size_t) const {
165                     return hash_numeric(key);
166                 }
167                 size_t operator()(std::string key, size_t) const {
168                     const char* c_key = key.c_str();
169                     return operator()(c_key, 0);
170                 }
171             };
172         }
173     }
174
175     template<typename T>
176     class GenericContainer {
177         /*
178             for the types we need to support other than const char* (ie int,
                    double, and std::string),
179             we can pass these around willy-nilly. for const char*, handled
                    below, we will obtain our
180             own copy of the character array by wrapping it in a std::string
181         */
182     private:
183         T raw;
184         functors::compare compare;
185     public:
186         GenericContainer(const T& val): raw(val) {}
187         GenericContainer() = default;
188         GenericContainer& operator=(GenericContainer const& rhs) = delete;
189         T operator()() const {
190             return raw;
```

```
191             }
192             T copy() const {
193                 return raw;
194             }
195             void reset(const T& val) {
196                 raw = val;
197             }
198             int compare_to(GenericContainer const& other) const {
199                 return compare(raw, other.raw);
200             }
201         };
202         template<>
203         class GenericContainer<const char*> {
204             /*
205                 class template specialization for character arrays, stores a local
                        copy of the character array
206             */
207         private:
208             char* raw = nullptr;
209             functors::compare compare;
210         public:
211             GenericContainer(const char* val) {
212                 reset(val);
213             }
214             GenericContainer() = default;
215             const char* operator()() const {
216                 return raw;
217             }
218             const char* copy() const {
219                 if (raw == nullptr) return nullptr;
220                 size_t len = strlen(raw);
221                 char* new_str = new char[len + 1];
222                 strncpy(new_str, raw, len);
223                 new_str[len] = 0;
224                 return new_str;
225             }
226             void reset(const char* val) {
227                 if (raw) {
228                     delete raw;
229                     raw = nullptr;
230                 }
231                 if (val != nullptr) {
232                     size_t len = strlen(val);
233                     raw = new char[len + 1];
234                     strncpy(raw, val, len);
235                     raw[len] = 0;
236                 }
237             }
238             int compare_to(GenericContainer const& other) const {
239                 return compare(raw, other.raw);
240             }
241         };
```

```
242
243         template<typename key_type,
244                  typename primary_hash =
                             hash_utils::functors::primary_hashes::hash_basic,
245                  typename secondary_hash =
                             hash_utils::functors::secondary_hashes::hash_double>
246         class Key {
247         private:
248             GenericContainer<key_type> raw_key;
249             primary_hash hasher1;
250             secondary_hash hasher2;
251             size_t hash1_val;
252             size_t hash2_val;
253         public:
254             Key& operator=(Key const& rhs) {
255                 if (&rhs == this)
256                     return *this;
257                 reset(rhs.raw_key());
258             }
259             bool operator==(Key const& rhs) const {
260                 return raw_key.compare_to(rhs.raw_key) == 0;
261             }
262             bool operator<(Key const& rhs) const {
263                 return raw_key.compare_to(rhs.raw_key) == -1;
264             }
265             bool operator>(Key const& rhs) const {
266                 return raw_key.compare_to(rhs.raw_key) == 1;
267             }
268             bool operator!=(Key const& rhs) const {
269                 return ! operator==(rhs);
270             }
271             size_t hash(size_t map_capacity, size_t probe_attempt) const {
272                 size_t local_hash2_val;
273                 if (probe_attempt != 0 && hasher2.changes_with_probe_attempt())
274                 {
275                     //if the hashing function value is dependent on the probe attempt
276                     //(eg quadratic probing), then we need to retrieve the new value*/
277                     local_hash2_val = hasher2(raw_key(), probe_attempt);
278                 } else {
279                     //otherwise we can just use the value we have stored
280                     local_hash2_val = hash2_val;
281                 }
282                 return (hash1_val + probe_attempt * local_hash2_val) % map_capacity;
283             }
284             key_type raw() const {
285                 return raw_key();
286             }
287             key_type raw_copy() const {
288                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
289                 return raw_key.copy();
290             }
```

```
291             template<typename T>
292             void reset(T key) {
293                 raw_key.reset(key);
294                 size_t base_probe_attempt = 0;
295                 hash1_val = hasher1(key);
296                 hash2_val = hasher2(key, base_probe_attempt);
297             }
298             void reset(const char* key) {
299                 raw_key.reset(key);
300                 if (key != nullptr) {
301                     size_t base_probe_attempt = 0;
302                     hash1_val = hasher1(key);
303                     hash2_val = hasher2(key, base_probe_attempt);
304                 }
305             }
306             explicit Key(key_type const& key): raw_key(key) {
307                 reset(key);
308             }
309             Key() = default;
310         };
311         template <typename value_type>
312         class Value {
313         private:
314             functors::compare compare;
315             GenericContainer<value_type> raw_value;
316         public:
317             Value& operator=(Value const& rhs) {
318                 if (&rhs == this)
319                     return *this;
320                 reset(rhs.raw_value());
321             }
322             bool operator==(Value const& rhs) const {
323                 return compare(raw_value(), rhs.raw_value());
324             }
325             bool operator==(value_type const& rhs) const {
326                 return compare(raw_value(), rhs) == 0;
327             }
328             value_type raw() const {
329                 return raw_value();
330             }
331             value_type raw_copy() const {
332                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
333                 return raw_value.copy();
334             }
335             void reset(value_type value) {
336                 raw_value.reset(value);
337             }
338             explicit Value(value_type const& value): raw_value(value) {}
339             Value() = default;
340         };
341     }
```

```
342 }
343
344 #endif
```

common/priority_queue.h

common/priority_queue.h

```
#ifndef _PRIORITY_QUEUE_H_
#define _PRIORITY_QUEUE_H_

#include "SDAL.h"
#include "common.h"

namespace cop3530 {
    //this class allows efficient sorting clusters by size for the
        cluster_distribution functions
    template<typename T,
            typename PriorityCompare =
                cop3530::hash_utils::ClusterInventory::cluster_size_less_predicate>
    class priority_queue {
    private:
        PriorityCompare first_arg_higher_priority;
        //SDAL has all the benefits of std::vector (ie fast random access and
            automatic resizing)
        //while having the added benefit of being legal to use in cop3530
        SDAL<T> tree;
        size_t num_items = 0;
        void fix_up(size_t index) {
            while (index > 1
                  && first_arg_higher_priority(tree[index], tree[index / 2]))
            {
                std::swap(tree[index / 2], tree[index]);
                index /= 2;
            }
        }
        void fix_down() {
            size_t parent_index = 1;
            while (2 * parent_index <= num_items) {
                size_t left_index = 2 * parent_index;
                size_t right_index = left_index + 1;
                size_t higher_priority_index = left_index;
                if (right_index <= num_items
                    && first_arg_higher_priority(tree[right_index], tree[left_index]))
                {
                    higher_priority_index = right_index;
                }
                if ( ! first_arg_higher_priority(tree[higher_priority_index],
                     tree[parent_index]))
                    //no more items to elevate
                    break;
                std::swap(tree[parent_index], tree[higher_priority_index]);
                parent_index = higher_priority_index;
            }
        }
```

```
44      public:
45          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
46              T empty_item;
47              tree.push_back(empty_item);
48          }
49          priority_queue(priority_queue const& src) {
50              tree = src.tree;
51              num_items = src.num_items;
52          }
53          T get_next_item() {
54              std::swap(tree[1], tree[num_items]);
55              T ret = tree[num_items--];
56              fix_down();
57              return ret;
58          }
59          void add_to_queue(T const& item) {
60              tree.push_back(item);
61              num_items++;
62              fix_up(num_items);
63          }
64          size_t size() {
65              return num_items;
66          }
67          bool empty() {
68              return num_items == 0;
69          }
70      };
71  }
72
73  #endif // _PRIORITY_QUEUE_H_
```

part3/bucket/source/buckets_map.h

part3/bucket/source/buckets_map.h

```
#ifndef _BUCKETS_MAP_GENERIC_H_
#define _BUCKETS_MAP_GENERIC_H_

#include <iostream>
#include "../../../common/common.h"
#include "../../../common/SSLL.h"
#include "../../../common/priority_queue.h"

namespace cop3530 {
    template<typename key_type,
             typename value_type,
             typename capacity_plan_functor =
                 hash_utils::functors::map_capacity_planner,
             typename hash = hash_utils::functors::primary_hashes::hash_basic>
    class HashMapBucketsGeneric {
    private:
        typedef hash_utils::ClusterInventory ClusterInventory;
        typedef hash_utils::Key<key_type, hash> Key;
        typedef hash_utils::Value<value_type> Value;
        struct Item {
            Key key;
            Value value;
            Item* next;
            bool is_dummy;
            explicit Item(Item* next): next(next), is_dummy(true) {}
        };
        struct Bucket {
            Item* head; //use a head pointer to the first node, and include a dummy
                node at the end (but dont store its pointer)
            Bucket() {
                Item* tail = new Item(nullptr);
                head = tail;
            }
            ~Bucket() {
                while ( ! head->is_dummy) {
                    Item* to_delete = head;
                    head = head->next;
                    delete to_delete;
                }
                delete head; //tail
            }
        };
        typedef Item* link;
        Bucket* buckets;
        size_t num_buckets = 0;
        size_t num_items = 0;
        /*
```

```
46              searches the bucket corresponding to the specified key's hash for that
47              key. if found, stores a reference to that item and returns P, the number
                    of
48              probe attempts needed to get to the item (ie the number of chain links
                    needed
49              to be traversed). otherwise return -1 * P and stores the pointer to the
                    tail dummy node in
50              item_ptr.
51          */
52          int search_internal(Key const& key, link& item_ptr) {
53              int probe_attempts = 1;
54              size_t hash_val = key.hash(capacity(), 0);
55              Bucket& bucket = buckets[hash_val];
56              item_ptr = bucket.head;
57              while ( ! item_ptr->is_dummy) {
58                  if (item_ptr->key == key) {
59                      //found the key
60                      return probe_attempts;
61                  }
62                  item_ptr = item_ptr->next;
63                  ++probe_attempts;
64              }
65              //key not found
66              return probe_attempts * -1;
67          }
68          void init() {
69              buckets = new Bucket[num_buckets];
70              num_items = 0;
71          }
72      public:
73          HashMapBucketsGeneric(size_t const min_buckets)
74          {
75              if (min_buckets == 0) {
76                  throw std::domain_error("min_buckets must be at least 1");
77              }
78              cop3530::hash_utils::functors::map_capacity_planner capacity_planner;
79              num_buckets = capacity_planner(min_buckets); //make capacity prime
80              init();
81          }
82          ~HashMapBucketsGeneric() {
83              delete[] buckets;
84          }
85          /*
86              if there is space available, adds the specified key/value-pair to the
                    hash map and returns the
87              number of probes required, P; otherwise returns -1 * P (that's a lie: we
                    will always have space
88              available because each bucket contains a linked list that is
                    indefinitely growable). If an item
89              already exists in the map with the same key, replace its value.
90          */
91          int insert(key_type const& key, value_type const& value) {
```

```
92              Item* item;
93              Key k(key);
94              Value v(value);
95              int probes_required = search_internal(k, item);
96              if (probes_required > 0)
97                  //found item
98                  item->value = v;
99              else {
100                 //currently holding tail (item not found). transform it into a valid
                        item then add a new tail
101                 item->is_dummy = false;
102                 item->key = k;
103                 item->value = v;
104                 item->next = new Item(nullptr);
105                 ++num_items;
106             }
107             return std::abs(probes_required);
108         }
109         /*
110             if there is an item matching key, removes the key/value-pair from the
                    map, stores it's value in
111             value, and returns the number of probes required, P; otherwise returns
                    -1 * P.
112         */
113         int remove(key_type const& key, value_type& value) {
114             Key k(key);
115             Item* item;
116             int probes_required = search_internal(k, item);
117             if (probes_required > 0) {
118                 //found item
119                 value = item->value.raw_copy();
120                 //swap the current item for the next one
121                 Item* to_delete = item->next;
122                 *item = *to_delete;
123                 delete to_delete;
124                 --num_items;
125             }
126             return probes_required;
127         }
128         /*
129             if there is an item matching key, stores it's value in value, and
                    returns the
130             number of probes required, P; otherwise returns -1 * P. Regardless, the
                    item
131             remains in the map.
132         */
133         int search(key_type const& key, value_type& value) {
134             Item* item;
135             Key k(key);
136             int probes_required = search_internal(k, item);
137             if (probes_required > 0) {
138                 //found item
```

```
139                 value = item->value.raw_copy();
140             }
141             return probes_required;
142         }
143         /*
144             removes all items from the map.
145         */
146         void clear() {
147             delete[] buckets;
148             init();
149         }
150         /*
151             returns true IFF the map contains no elements.
152         */
153         bool is_empty() {
154             return size() == 0;
155         }
156         /*
157             returns the number of slots in the map.
158         */
159         size_t capacity() {
160             return num_buckets;
161         }
162         /*
163             returns the number of items actually stored in the map.
164         */
165         size_t size() {
166             return num_items;
167         }
168         /*
169             returns the map's load factor (occupied buckets = load * capacity).
170         */
171         double load() {
172             size_t occupied_buckets = 0;
173             if (size() > 0) {
174                 size_t M = capacity();
175                 for (size_t i = 0; i != M; ++i) {
176                     Bucket const& bucket = buckets[i];
177                     if ( ! bucket.head->is_dummy)
178                         //bucket has at least one item
179                         occupied_buckets++;
180                 }
181             }
182             return static_cast<double>(occupied_buckets) / capacity();
183         }
184         /*
185             inserts into the ostream, the backing array's contents in sequential
                order.
186             Empty slots shall be denoted by a hyphen, non-empty slots by that item's
187             key. [This function will be used for debugging/monitoring].
188         */
189         std::ostream& print(std::ostream& out) {
```

```
190             size_t cap = capacity();
191             bool print_separator = false;
192             out << '[';
193             for (size_t i = 0; i != cap; ++i) {
194                 Bucket const& bucket = buckets[i];
195                 if (bucket.head->is_dummy) {
196                     if (print_separator)
197                         out << "|";
198                     else
199                         print_separator = true;
200                     out << "-";
201                 } else {
202                     for (Item* item = bucket.head; item->is_dummy != true; item =
                         item->next) {
203                         if (print_separator)
204                             out << "|";
205                         else
206                             print_separator = true;
207                         out << item->key.raw();
208                     }
209                 }
210             }
211             out << ']';
212             return out;
213         }
214
215         /*
216             returns a priority queue containing cluster sizes and instances (in the
                form of ClusterInventory
217             struct instances), sorted by cluster size.
218         */
219         priority_queue<ClusterInventory> cluster_distribution() {
220             //use a simple linked list to count cluster instances, then feed those
                to a priority queue and return it.
221             priority_queue<ClusterInventory> cluster_pq;
222             if (size() == 0) return cluster_pq;
223             SSLL<ClusterInventory> clusters;
224             size_t M = capacity();
225             for (size_t i = 0; i != M; ++i) {
226                 Bucket const& bucket = buckets[i];
227                 size_t bucket_size = 0;
228                 Item* item_ptr = bucket.head;
229                 while ( ! item_ptr->is_dummy) {
230                     ++bucket_size;
231                     item_ptr = item_ptr->next;
232                 }
233                 //I don't love this O(N^2) implementation, but premature
                    optimization is the root of all evil and late projects
234                 SSLL<ClusterInventory>::iterator cluster_iterator = clusters.begin();
235                 SSLL<ClusterInventory>::iterator cluster_iterator_end =
                    clusters.end();
236                 bool found_cluster = false;
```

```
237                 for (; cluster_iterator != cluster_iterator_end; ++cluster_iterator)
                        {
238                     if (cluster_iterator->cluster_size == bucket_size) {
239                         found_cluster = true;
240                         break;
241                     }
242                 }
243                 if (found_cluster)
244                     cluster_iterator->num_instances++;
245                 else
246                     clusters.push_back({bucket_size, 1});
247             }
248             SSLL<ClusterInventory>::const_iterator cluster_iterator =
                    clusters.begin();
249             SSLL<ClusterInventory>::const_iterator cluster_iterator_end =
                    clusters.end();
250             for (; cluster_iterator != cluster_iterator_end; ++cluster_iterator) {
251                 if (cluster_iterator->cluster_size > 0)
252                     cluster_pq.add_to_queue(*cluster_iterator);
253             }
254             return cluster_pq;
255         }
256
257         /*
258             generate a random number, R, (1,size), and starting with slot zero in
                    the backing array,
259             find the R-th occupied slot; remove the item from that slot (adjusting
                    subsequent items as
260             necessary), and return its key.
261         */
262         key_type remove_random() {
263             if (size() == 0) throw std::logic_error("Cant remove from an empty map");
264             size_t num_slots = capacity();
265             size_t ith_node_to_delete = 1 + rand_i(size());
266             for (size_t i = 0; i != num_slots; ++i) {
267                 Bucket const& bucket = buckets[i];
268                 Item* item_ptr = bucket.head;
269                 while ( ! item_ptr->is_dummy) {
270                     if (--ith_node_to_delete == 0) {
271                         key_type key = item_ptr->key.raw_copy();
272                         value_type val_dummy;
273                         remove(key, val_dummy);
274                         return key;
275                     }
276                     item_ptr = item_ptr->next;
277                 }
278             }
279             throw std::logic_error("Unexpected end of remove_random function");
280         }
281     };
282 }
283
```

```
284 #endif
```

Part IV: Randomized BST

Part 4 RBST Testing Strategy

Paul Nickerson

operation_failures.cpp

Within the failure operations scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start with an empty map and try to search() and remove() an item whose key does not exist in the map. Both calls should return a value less than zero (indicating key not found). I then fill the map with a bunch of items (it is impossible to run out of space because collisions are resolved via an arbitrarily-growable liked list). From this newly-filled map, I attempt to remove() a key that doesn't exist in the map, and search() for a key that does not exist in the map, both of which should return a value less than zero.

The RBST class keeps track of the height and number of children of each subtree with O(1) complexity during each operation that potentially changes those values. Because these values are so crucial to the map's functionality, and considering the difficulty of externally validating those values without adding more public methods, I include a preprocessor macro, _DEBUG_, which, when set to true (it defaults to false in production uses of the class), indicates to the BST base class to recursively verify the value of these values before each public method returns. This is potentially an expensive O(N) operation that significantly slows down map functionality, which is why it is disabled by default and specifically enabled during unit testing. If it is determined that the O(1) calculated values do not match the O(N), "true" values, the class throws an exception, which is caught by the CATCH testing framework and causes the test to fail.

operation_successes.cpp

Within the success operations scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start by filling the map with a bunch of items, clearing it, then filling it up again. The map should then report the correct size. I check that several keys which are expected to exist in the map actually do exist (including the lowest possible key, the highest possible key, and one in the middle).

I check the print() function by routing it to an output string stream and count the number of hyphens in the output, which indicate empty slots. Since load factor = occupied buckets / capacity, we can get the number of unoccupied buckets as capacity * (1 - load). This should equal the number of hyphens in the print() output.

I then attempt to remove() several keys which are known to exist, and check that their associated values are what were expected. After these items are removed, I try to both search() and remove() them, which should all return false.

I test remove_random() in a similar way to the preceeding remove() check. I remove a random key, then try to search() for it and remove() it. Both checks should fail and return a value less than zero.

tree_structure.cpp

Because I was able to successfully implement the pretty-print bonus method, it is fairly straightforward to verify that the underlying RBST map successfully maintains the correct BST tree structure. I print the tree structure to a file, then make a system() call to pipe that through a few command line utilities that extract just the numerical key (I say numerical, but that number could be in the form of a string of const char*). This results in the natural order of the keys as they exist in the tree. I pipe those through uniq and sort to remove duplicates and induce expected sorting order. If the tree exhibits the correct structure, the output of those two operations will match verbatim.

I start by filling the map to half capacity, verify that load() returns 0.5, then check validate the tree structure. Afterwards, I fill the tree to full capacity and then delete half the nodes via remove() and remove_random() operations in an attempt to destablize tree structure. Then I validate the tree structure again.

This series of checks is implemented for each of the four supported key types.

part4/part4.pdf

part4/checklist.txt

```
Randomized BST written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part IV: Randomized BST
=============================================================================
My MAP implementation uses the data structure described in the part IV
instructions and conforms to the technique required for this map
variety: yes

My MAP implementation 100% correctly implements RBST behavior: yes

My MAP implementation 100% correctly supports the following key types:
* signed int: yes
* double: yes
* c-string: yes
* std::string: yes

My MAP implementation 100% correctly supports the ANY value type: yes

My MAP implementation 100% correctly supports the following methods
as described in part IV:

* insert: yes
* remove: yes
* search: yes
* clear: yes
* is_empty: yes
* capacity: yes
* size: yes
* load: yes
* print: yes
* cluster_distribution(): yes
* remove_random(): yes

My MAP implementation 100% correctly implements the bonus print(): yes

=============================================================================
FOR ALL PARTS
=============================================================================

My MAP implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes
```

```
My TEST compiles correctly using g++ v4.8.2 on the OpenBSD VM: yes

My TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this Randomized BST
and the associated tests.
Paul Nickerson, Dec 2 in COP3520 section 1087

------------------------------------------------------------------------
------------------------------------------------------------------------

How to compile and run my unit tests on the OpenBSD VM
cd part4/source
./compile.sh
./run_tests > output.txt
```

common/common.h

common/common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#ifndef _DEBUG_
    //various internal integrity checks that can be expensive, we want to disable them in production
    #define _DEBUG_ false
#endif

#include <string.h>
#include <limits>
#include <stdexcept>
#include <ostream>
#include <cmath>

namespace cop3530 {
    inline double lg(size_t i) {
        return std::log(i) / std::log(2);
    }
    inline size_t rand_i(size_t max) {
        size_t bucket_size = RAND_MAX / max;
        size_t num_buckets = RAND_MAX / bucket_size;
        size_t big_rand;
        do {
               big_rand = rand();
        } while(big_rand >= num_buckets * bucket_size);
        return big_rand / bucket_size;
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        static constexpr size_t primes[] = { //from algorithms in c++, helps us to choose a prime-number map capacity
            251,
            509,
            1021,
            2039,
            4093,
            8191,
            16381,
            32749,
            65521,
            131071,
            262193,
            524287,
            1048573,
            2097143,
```

```
            4194301,
            8388593,
            16777213,
            33554393,
            67108859,
            134217689,
            268435399,
            536870909,
            1073741789,
            2147483647
        };
        struct ClusterInventory {
            size_t cluster_size;
            size_t num_instances;
            struct cluster_size_less_predicate {
                bool operator()(ClusterInventory const& cluster1, ClusterInventory
                     const& cluster2) {
                    return cluster1.cluster_size < cluster2.cluster_size;
                }
            };
        };

        inline size_t str_to_numeric(const char* str) {
            unsigned int base = 257; //prime number chosen near an 8-bit character
            size_t numeric = 0;
            for (; *str != 0; ++str)
                numeric = numeric * base + *str;
            return numeric;
        }
        namespace functors {

            struct map_capacity_planner {
                size_t operator()(size_t min_capacity) {
                    for (int i = 0; i != 24; ++i)
                        if (min_capacity < primes[i])
                            return primes[i];
                    throw std::domain_error("Provided min capacity too large.
                         Consider extending the list of prime numbers");
                }
            };
            struct compare {
                int operator()(const char* a, const char* b) const {
                    int cmp = strcmp(a, b);
                    return (cmp < 0 ? -1 :
                                    (cmp > 0 ? 1 : 0));
                }
                int operator()(double a, double b) const {
                    return (a < b ? -1 :
                                  (a > b ? 1 : 0));
                }
                int operator()(std::string const& a, std::string const& b) const {
                    return (a < b ? -1 :
```

```
                        (a > b ? 1 : 0));
    }
    int operator()(int a, int b) const {
        return (a < b ? -1 :
                        (a > b ? 1 : 0));
    }
};
namespace primary_hashes {
    struct hash_basic {
    //this is such a stupid hash method, but unlike my pathetic attempts at implementing
    //various other hashing methods, it works and is generalizable to all the required key
    //types. together with double hashing it should make for a passable hashing routine.
    public:
        size_t operator()(const char* key) const {
            return str_to_numeric(key);
        }
        size_t operator()(double key) const {
            return static_cast<size_t>(std::fmod(key, max_size_t));
        }
        size_t operator()(int key) const {
            return static_cast<size_t>(key);
        }
        size_t operator()(std::string const& key) const {
            const char* c_key = key.c_str();
            return operator()(c_key);
        }
    };
}
namespace secondary_hashes {
    struct linear_probe {
        bool changes_with_probe_attempt() const {
            return false;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return 1;
        }
    };
    struct quadratic_probe {
        bool changes_with_probe_attempt() const {
            return true;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return probe_attempt;
        }
    };
    struct hash_double {
    private:
```

```
145                 size_t hash_numeric(size_t numeric) const {
146                     size_t hash = numeric % 97; //simple modulus using a prime
                            number (from algorithms in c++)
147                     //the second hash may not be zero (will cause an infinite
                            loop).
148                     //also, hash must be relatively prime to map_capacity so that
                            every slot can be hit.
149                     //map capacity is a prime number based chosen from the table,
                            so any value less than
150                     //map capacity should work
151                     return hash;
152                 }
153             public:
154                 bool changes_with_probe_attempt() const {
155                     return false;
156                 }
157                 size_t operator()(const char* key, size_t) const {
158                     size_t numeric = str_to_numeric(key);
159                     return hash_numeric(numeric);
160                 }
161                 size_t operator()(double key, size_t) const {
162                     return hash_numeric(key);
163                 }
164                 size_t operator()(int key, size_t) const {
165                     return hash_numeric(key);
166                 }
167                 size_t operator()(std::string key, size_t) const {
168                     const char* c_key = key.c_str();
169                     return operator()(c_key, 0);
170                 }
171             };
172         }
173     }
174
175     template<typename T>
176     class GenericContainer {
177         /*
178             for the types we need to support other than const char* (ie int,
                    double, and std::string),
179             we can pass these around willy-nilly. for const char*, handled
                    below, we will obtain our
180             own copy of the character array by wrapping it in a std::string
181         */
182     private:
183         T raw;
184         functors::compare compare;
185     public:
186         GenericContainer(const T& val): raw(val) {}
187         GenericContainer() = default;
188         GenericContainer& operator=(GenericContainer const& rhs) = delete;
189         T operator()() const {
190             return raw;
```

```
191             }
192             T copy() const {
193                 return raw;
194             }
195             void reset(const T& val) {
196                 raw = val;
197             }
198             int compare_to(GenericContainer const& other) const {
199                 return compare(raw, other.raw);
200             }
201         };
202         template<>
203         class GenericContainer<const char*> {
204             /*
205                 class template specialization for character arrays, stores a local
                        copy of the character array
206             */
207         private:
208             char* raw = nullptr;
209             functors::compare compare;
210         public:
211             GenericContainer(const char* val) {
212                 reset(val);
213             }
214             GenericContainer() = default;
215             const char* operator()() const {
216                 return raw;
217             }
218             const char* copy() const {
219                 if (raw == nullptr) return nullptr;
220                 size_t len = strlen(raw);
221                 char* new_str = new char[len + 1];
222                 strncpy(new_str, raw, len);
223                 new_str[len] = 0;
224                 return new_str;
225             }
226             void reset(const char* val) {
227                 if (raw) {
228                     delete raw;
229                     raw = nullptr;
230                 }
231                 if (val != nullptr) {
232                     size_t len = strlen(val);
233                     raw = new char[len + 1];
234                     strncpy(raw, val, len);
235                     raw[len] = 0;
236                 }
237             }
238             int compare_to(GenericContainer const& other) const {
239                 return compare(raw, other.raw);
240             }
241         };
```

```
242
243         template<typename key_type,
244                  typename primary_hash =
                         hash_utils::functors::primary_hashes::hash_basic,
245                  typename secondary_hash =
                         hash_utils::functors::secondary_hashes::hash_double>
246         class Key {
247         private:
248             GenericContainer<key_type> raw_key;
249             primary_hash hasher1;
250             secondary_hash hasher2;
251             size_t hash1_val;
252             size_t hash2_val;
253         public:
254             Key& operator=(Key const& rhs) {
255                 if (&rhs == this)
256                     return *this;
257                 reset(rhs.raw_key());
258             }
259             bool operator==(Key const& rhs) const {
260                 return raw_key.compare_to(rhs.raw_key) == 0;
261             }
262             bool operator<(Key const& rhs) const {
263                 return raw_key.compare_to(rhs.raw_key) == -1;
264             }
265             bool operator>(Key const& rhs) const {
266                 return raw_key.compare_to(rhs.raw_key) == 1;
267             }
268             bool operator!=(Key const& rhs) const {
269                 return ! operator==(rhs);
270             }
271             size_t hash(size_t map_capacity, size_t probe_attempt) const {
272                 size_t local_hash2_val;
273                 if (probe_attempt != 0 && hasher2.changes_with_probe_attempt())
274                 {
275                     //if the hashing function value is dependent on the probe attempt
276                     //(eg quadratic probing), then we need to retrieve the new value*/
277                     local_hash2_val = hasher2(raw_key(), probe_attempt);
278                 } else {
279                     //otherwise we can just use the value we have stored
280                     local_hash2_val = hash2_val;
281                 }
282                 return (hash1_val + probe_attempt * local_hash2_val) % map_capacity;
283             }
284             key_type raw() const {
285                 return raw_key();
286             }
287             key_type raw_copy() const {
288                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
289                 return raw_key.copy();
290             }
```

```
291             template<typename T>
292             void reset(T key) {
293                 raw_key.reset(key);
294                 size_t base_probe_attempt = 0;
295                 hash1_val = hasher1(key);
296                 hash2_val = hasher2(key, base_probe_attempt);
297             }
298             void reset(const char* key) {
299                 raw_key.reset(key);
300                 if (key != nullptr) {
301                     size_t base_probe_attempt = 0;
302                     hash1_val = hasher1(key);
303                     hash2_val = hasher2(key, base_probe_attempt);
304                 }
305             }
306             explicit Key(key_type const& key): raw_key(key) {
307                 reset(key);
308             }
309             Key() = default;
310         };
311         template <typename value_type>
312         class Value {
313         private:
314             functors::compare compare;
315             GenericContainer<value_type> raw_value;
316         public:
317             Value& operator=(Value const& rhs) {
318                 if (&rhs == this)
319                     return *this;
320                 reset(rhs.raw_value());
321             }
322             bool operator==(Value const& rhs) const {
323                 return compare(raw_value(), rhs.raw_value());
324             }
325             bool operator==(value_type const& rhs) const {
326                 return compare(raw_value(), rhs) == 0;
327             }
328             value_type raw() const {
329                 return raw_value();
330             }
331             value_type raw_copy() const {
332                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
333                 return raw_value.copy();
334             }
335             void reset(value_type value) {
336                 raw_value.reset(value);
337             }
338             explicit Value(value_type const& value): raw_value(value) {}
339             Value() = default;
340         };
341     }
```

```
342 }
343
344 #endif
```

common/priority_queue.h

common/priority_queue.h

```
#ifndef _PRIORITY_QUEUE_H_
#define _PRIORITY_QUEUE_H_

#include "SDAL.h"
#include "common.h"

namespace cop3530 {
    //this class allows efficient sorting clusters by size for the
        cluster_distribution functions
    template<typename T,
            typename PriorityCompare =
                cop3530::hash_utils::ClusterInventory::cluster_size_less_predicate>
    class priority_queue {
    private:
        PriorityCompare first_arg_higher_priority;
        //SDAL has all the benefits of std::vector (ie fast random access and
            automatic resizing)
        //while having the added benefit of being legal to use in cop3530
        SDAL<T> tree;
        size_t num_items = 0;
        void fix_up(size_t index) {
            while (index > 1
                  && first_arg_higher_priority(tree[index], tree[index / 2]))
            {
                std::swap(tree[index / 2], tree[index]);
                index /= 2;
            }
        }
        void fix_down() {
            size_t parent_index = 1;
            while (2 * parent_index <= num_items) {
                size_t left_index = 2 * parent_index;
                size_t right_index = left_index + 1;
                size_t higher_priority_index = left_index;
                if (right_index <= num_items
                    && first_arg_higher_priority(tree[right_index], tree[left_index]))
                {
                    higher_priority_index = right_index;
                }
                if ( ! first_arg_higher_priority(tree[higher_priority_index],
                     tree[parent_index]))
                    //no more items to elevate
                    break;
                std::swap(tree[parent_index], tree[higher_priority_index]);
                parent_index = higher_priority_index;
            }
        }
```

```
44      public:
45          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
46              T empty_item;
47              tree.push_back(empty_item);
48          }
49          priority_queue(priority_queue const& src) {
50              tree = src.tree;
51              num_items = src.num_items;
52          }
53          T get_next_item() {
54              std::swap(tree[1], tree[num_items]);
55              T ret = tree[num_items--];
56              fix_down();
57              return ret;
58          }
59          void add_to_queue(T const& item) {
60              tree.push_back(item);
61              num_items++;
62              fix_up(num_items);
63          }
64          size_t size() {
65              return num_items;
66          }
67          bool empty() {
68              return num_items == 0;
69          }
70      };
71  }
72
73  #endif // _PRIORITY_QUEUE_H_
```

common/unit_test_utils.h

common/unit_test_utils.h

```
#ifndef _UNIT_TEST_UTILS_H_
#define _UNIT_TEST_UTILS_H_

#include <iostream>
#include <string>
#include <fstream>

namespace cop3530 {
    namespace unit_test_utils {
        inline std::string guid() {
            std::string ret = "";
            for (size_t i = 0; i != 32; ++i) {
                size_t rnd = cop3530::rand_i(16);
                if (rnd < 10)
                    ret += std::string(1, '0' + rnd);
                else
                    ret += std::string(1, 'A' + rnd - 10);
            }
            return ret;
        }
        inline std::string get_tmp_filename() {
            return std::string("/tmp/") + guid() + std::string(".out");
        }
        inline void delete_file(std::string file_path) {
            system((std::string("rm ") + file_path + std::string(" 2>&1 >>
                /tmp/debug")).c_str());
        }
        inline std::string shell_cmd(std::string cmd) {
            std::string shell_script_file = get_tmp_filename();
            std::string output_file = get_tmp_filename();
            std::ofstream shell_script_out(shell_script_file);
            shell_script_out << "#!/bin/sh" << std::endl << cmd << std::endl;
            shell_script_out.close();
            std::string chmod_cmd = std::string("chmod +x ") + shell_script_file;
            system(chmod_cmd.c_str());
            std::string invoke_cmd = shell_script_file + std::string(" > ") +
                output_file;
            system(invoke_cmd.c_str());
            std::ifstream read_output(output_file);
            std::ostringstream oss;
            std::string tmp;
            while (std::getline(read_output, tmp)) {
                oss << tmp;
                if ( ! read_output.eof())
                    oss << "\n";
            }
            delete_file(output_file);
```

```
46              delete_file(shell_script_file);
47              return oss.str();
48          }
49      }
50  }
51
52  #endif // _UNIT_TEST_UTILS_H_
```

part4/source/bst.h

part4/source/bst.h

```
#ifndef _BST_H_
#define _BST_H_

#include <cstdlib>
#include <sstream>
#include "../../common/CDAL.h"
#include "../../common/common.h"
#include "../../common/priority_queue.h"

namespace cop3530 {
    template<typename key_type,
             typename value_type>
    class BST {
    protected: //let RBST and AVL inherit everything
        typedef hash_utils::ClusterInventory ClusterInventory;
        typedef hash_utils::Key<key_type> Key;
        typedef hash_utils::Value<value_type> Value;
        struct Node;
        typedef Node* link;
        struct Node {
            Key key;
            Value value;
            size_t num_children;
            size_t left_index;
            size_t right_index;
            size_t height; //height tracking coded in this class, but not used (for
                AVL, which is this class with self-balancing)
            bool is_occupied;
            size_t validate_children_count_recursive(Node* nodes) {
                //this function is for debugging purposes, does recursive traversal
                    to find the correct number of children
                size_t child_count = 0;
                if (left_index)
                    child_count += 1 +
                        nodes[left_index].validate_children_count_recursive(nodes);
                if (right_index)
                    child_count += 1 +
                        nodes[right_index].validate_children_count_recursive(nodes);
                if (child_count != num_children) {
                    std::ostringstream msg;
                    msg << "Manually counted children, " << child_count << ",
                        different than child count, " << num_children;
                    throw std::logic_error(msg.str());
                }
                return child_count;
            }
            size_t get_height_recursive(Node* nodes) {
```

```
43              //this function is for debugging purposes, does recursive traversal
                    to find the correct height
44              size_t left_height = 0, right_height = 0;
45              size_t calculated_height = 0;
46              if (left_index)
47                  left_height = nodes[left_index].get_height_recursive(nodes);
48              if (right_index)
49                  right_height = nodes[right_index].get_height_recursive(nodes);
50              calculated_height = 1 + std::max(left_height, right_height);
51              return calculated_height;
52          }
53          void update_height(Node* nodes) {
54              //note: this method depends on the left and right subtree heights
                    being correct
55              size_t left_height = 0, right_height = 0;
56              if (left_index)
57                  left_height = nodes[left_index].height;
58              if (right_index)
59                  right_height = nodes[right_index].height;
60              height = 1 + std::max(left_height, right_height);
61              if (_DEBUG_) {
62                  size_t calculated_height = get_height_recursive(nodes);
63                  if (calculated_height != height) {
64                      std::ostringstream msg;
65                      msg << "Manually calculated height, " << calculated_height <<
                            ", different than tracked height, " << height;
66                      throw std::logic_error(msg.str());
67                  }
68              }
69          }
70          void disable_and_adopt_free_tree(size_t free_index) {
71              is_occupied = false;
72              height = 0;
73              num_children = 0;
74              right_index = 0;
75              left_index = free_index;
76          }
77          void reset_and_enable(Key const& new_key, Value const& new_value) {
78              is_occupied = true;
79              height = 1; //self
80              left_index = right_index = 0;
81              num_children = 0;
82              key = new_key;
83              value = new_value;
84          }
85          int balance_factor(const Node* nodes) const {
86              size_t left_height = 0, right_height = 0;
87              if (left_index)
88                  left_height = nodes[left_index].height;
89              if (right_index)
90                  right_height = nodes[right_index].height;
```

```
91              return static_cast<long int>(left_height) - static_cast<long
                    int>(right_height);
92          }
93      };
94      Node* nodes; //***note: array is 1-based so leaf nodes have child indices
            set to zero
95      size_t free_index;
96      size_t root_index;
97      size_t curr_capacity;
98      virtual size_t remove_smallest_key_node_index(size_t& subtree_root_index) {
99          //returns the index of the node with the smallest key, while
100         //setting its parent's left child index to the smallest key node's
101         //right child index. recursion downward through this function updates
102         //the heights of the nodes it traverses
103         Node& subtree_root = nodes[subtree_root_index];
104         size_t smallest_key_node_index = 0;
105         if (subtree_root_index == 0) {
106             throw std::logic_error("Expected to find a valid node, but didn't");
107         } else {
108             if (subtree_root.left_index) {
109                 smallest_key_node_index =
                        remove_smallest_key_node_index(subtree_root.left_index);
110                 subtree_root.num_children--;
111                 subtree_root.update_height(nodes);
112             } else {
113                 smallest_key_node_index = subtree_root_index;
114                 subtree_root_index = subtree_root.right_index;
115             }
116         }
117         return smallest_key_node_index;
118     }
119     virtual size_t remove_largest_key_node_index(size_t& subtree_root_index) {
120         //returns the index of the node with the largest key, while
121         //setting its parent's right child index to the largest key node's
122         //left child index. recursion downward through this function updates
123         //the heights of the nodes it traverses
124         Node& subtree_root = nodes[subtree_root_index];
125         size_t largest_key_node_index = 0;
126         if (subtree_root_index == 0) {
127             throw std::logic_error("Expected to find a valid node, but didn't");
128         } else {
129             if (subtree_root.right_index) {
130                 largest_key_node_index =
                        remove_largest_key_node_index(subtree_root.right_index);
131                 subtree_root.num_children--;
132                 subtree_root.update_height(nodes);
133             } else {
134                 largest_key_node_index = subtree_root_index;
135                 subtree_root_index = subtree_root.left_index;
136             }
137         }
138         return largest_key_node_index;
```

```
139         }
140         virtual void remove_node(size_t& subtree_root_index) {
141             Node& subtree_root = nodes[subtree_root_index];
142             size_t index_to_delete = subtree_root_index;
143             if (subtree_root.right_index || subtree_root.left_index) {
144                 //subtree has at least one child
145                 if (subtree_root.right_index)
146                     //replace the root with the smallest-keyed node in the right
                            subtree
147                     subtree_root_index =
                            remove_smallest_key_node_index(subtree_root.right_index);
148                 else if (subtree_root.left_index)
149                     //replace the root with the largest-keyed node in the left subtree
150                     subtree_root_index =
                            remove_largest_key_node_index(subtree_root.left_index);
151                 //have the new root adopt the old root's children
152                 Node& new_root = nodes[subtree_root_index];
153                 new_root.left_index = subtree_root.left_index;
154                 new_root.right_index = subtree_root.right_index;
155                 //the new root has the same number of children as the old root,
                        minus one
156                 new_root.num_children = subtree_root.num_children - 1;
157                 //removing the smallest/largest-keyed node from the old root has the
                        effect of
158                 //updating the heights of the old root's relevant subtrees (which
                        the new root
159                 //just adopted), so we can update the new root's height now
160                 new_root.update_height(nodes);
161             } else
162                 //neither subtree exists, so just delete the node
163                 subtree_root_index = 0;
164             //node has been disowned by all ancestors, and has disowned all
                    descendents, so free it
165             add_node_to_free_tree(index_to_delete);
166         }
167         virtual int do_remove(size_t nodes_visited, //starts at 0 when this
                function is first called (ie does not include current node visitation)
168                               size_t& subtree_root_index,
169                               Key const& key,
170                               Value& value,
171                               bool& found_key)
172         {
173             if (subtree_root_index != 0) {
174                 Node& subtree_root = nodes[subtree_root_index];
175                 ++nodes_visited;
176                 //keep going down to the base of the tree
177                 if (key < subtree_root.key) {
178                     nodes_visited = do_remove(nodes_visited, subtree_root.left_index,
                            key, value, found_key);
179                     if (found_key) {
180                         //found the desired node and delete it
181                         subtree_root.num_children--;
```

```
                    //left child changed, so recompute subtree height
                    subtree_root.update_height(nodes);
                }
            } else if (key > subtree_root.key) {
                nodes_visited = do_remove(nodes_visited,
                     subtree_root.right_index, key, value, found_key);
                if (found_key) {
                    //found the desired node and delete it
                    subtree_root.num_children--;
                    //right child changed, so recompute subtree height
                    subtree_root.update_height(nodes);
                }
            } else if (key == subtree_root.key) {
                //found key, remove the node
                found_key = true;
                value = subtree_root.value;
                remove_node(subtree_root_index);
            } else {
                throw std::logic_error("Unexpected compare result");
            }
        }
        return nodes_visited;
    }
    void write_subtree_buffer(size_t subtree_root_index,
                          CDAL<std::string>& buffer_lines,
                          size_t root_line_index,
                          size_t lbound_line_index /*inclusive*/,
                          size_t ubound_line_index /*exclusive*/) const
    {
        Node subtree_root = nodes[subtree_root_index];
        std::ostringstream oss;
        //print the node
        //todo: fix this to only print the key
        oss << "[" << subtree_root.key.raw() << "]";
        buffer_lines[root_line_index] += oss.str();
        //print the right descendents
        if (subtree_root.right_index > 0) {
            //at least 1 right child
            size_t top_dashes = 1;
            Node const& right_child = nodes[subtree_root.right_index];
            if (right_child.left_index > 0) {
                //right child has at least 1 left child
                Node const& right_left_child = nodes[right_child.left_index];
                top_dashes += 2 * (1 + right_left_child.num_children);
            }
            size_t top_line_index = root_line_index - 1;
            while (top_line_index >= root_line_index - top_dashes)
                buffer_lines[top_line_index--] += "| ";
            size_t right_child_line_index = top_line_index;
            buffer_lines[top_line_index--] += "+--";
            while (top_line_index >= lbound_line_index)
                buffer_lines[top_line_index--] += "  ";
```

```
233             write_subtree_buffer(subtree_root.right_index,
234                                  buffer_lines,
235                                  right_child_line_index,
236                                  lbound_line_index,
237                                  root_line_index);
238         }
239         //print the left descendents
240         if (subtree_root.left_index > 0) {
241             //at least 1 left child
242             size_t bottom_dashes = 1;
243             Node const& left_child = nodes[subtree_root.left_index];
244             if (left_child.right_index > 0) {
245                 //left child has at least 1 right child
246                 Node const& left_right_child = nodes[left_child.right_index];
247                 bottom_dashes += 2 * (1 + left_right_child.num_children);
248             }
249             size_t bottom_line_index = root_line_index + 1;
250             while (bottom_line_index <= root_line_index + bottom_dashes)
251                 buffer_lines[bottom_line_index++] += "| ";
252             size_t left_child_line_index = bottom_line_index;
253             buffer_lines[bottom_line_index++] += "+--";
254             while (bottom_line_index < ubound_line_index)
255                 buffer_lines[bottom_line_index++] += " ";
256             write_subtree_buffer(subtree_root.left_index,
257                                  buffer_lines,
258                                  left_child_line_index,
259                                  root_line_index + 1,
260                                  ubound_line_index);
261         }
262     }
263     void add_node_to_free_tree(size_t node_index) {
264         nodes[node_index].disable_and_adopt_free_tree(free_index);
265         free_index = node_index;
266     }
267     size_t procure_node(Key const& key, Value const& value) {
268         //updates the free index to the first free node's left child (while
                transforming that first free
269         //node to an enabled node with the specified key/value) and returns the
                index of what was the last
270         //free index
271         size_t node_index = free_index;
272         free_index = nodes[free_index].left_index;
273         Node& n = nodes[node_index];
274         n.reset_and_enable(key, value);
275         return node_index;
276     }
277     virtual int insert_at_leaf(size_t nodes_visited, //starts at 0 when this
            function is first called (ie does not include current node visitation)
278                                size_t& subtree_root_index,
279                                Key const& key,
280                                Value const& value,
281                                bool& found_key)
```

```
282         {
283             if (subtree_root_index == 0) {
284                 //key not found
285                 subtree_root_index = procure_node(key, value);
286             } else {
287                 //parent was not a leaf
288                 //keep going down to the base of the tree
289                 Node& subtree_root = nodes[subtree_root_index];
290                 ++nodes_visited;
291                 if (key < subtree_root.key) {
292                     nodes_visited = insert_at_leaf(nodes_visited,
                            subtree_root.left_index, key, value, found_key);
293                     if ( ! found_key) {
294                         //given key is unique to the tree, so a new node was added
295                         subtree_root.num_children++;
296                         subtree_root.update_height(nodes);
297                     }
298                 } else if (key > subtree_root.key) {
299                     nodes_visited = insert_at_leaf(nodes_visited,
                            subtree_root.right_index, key, value, found_key);
300                     if ( ! found_key) {
301                         //given key is unique to the tree, so a new node was added
302                         subtree_root.num_children++;
303                         subtree_root.update_height(nodes);
304                     }
305                 } else if (key == subtree_root.key) {
306                     //found key, replace the value
307                     subtree_root.value = value;
308                     found_key = true;
309                 } else {
310                     throw std::logic_error("Unexpected compare result");
311                 }
312             }
313             return nodes_visited;
314         }
315         void rotate_left(size_t& subtree_root_index) {
316             Node& subtree_root = nodes[subtree_root_index];
317             size_t right_child_index = subtree_root.right_index;
318             Node& right_child = nodes[right_child_index];
319
320             //original root adopts the right child's left subtree
321             subtree_root.right_index = right_child.left_index;
322             //original root adopted a subtree (whose height did not change), so
                    update its height
323             subtree_root.update_height(nodes);
324
325             //right child adopts original root and its children
326             right_child.left_index = subtree_root_index;
327             //right child (new root) adopted the original root (whose height has
                    been updated), so update its height
328             right_child.update_height(nodes);
```

```
329             //since right child took the subtree root's place, it has the same
                    number of children as the original root
330             right_child.num_children = subtree_root.num_children;
331
332             //root has new children, so update that counter (done after changing the
                    right child's children counter
333             //because that depends on the original root's counter)
334             subtree_root.num_children = 0;
335             if (subtree_root.left_index != 0)
336                 subtree_root.num_children += 1 +
                        nodes[subtree_root.left_index].num_children;
337             if (subtree_root.right_index != 0)
338                 subtree_root.num_children += 1 +
                        nodes[subtree_root.right_index].num_children;
339
340             //set the right child as the new root
341             subtree_root_index = right_child_index;
342         }
343         void rotate_right(size_t& subtree_root_index) {
344             Node& subtree_root = nodes[subtree_root_index];
345             size_t left_child_index = subtree_root.left_index;
346             Node& left_child = nodes[left_child_index];
347
348             //original root adopts the left child's right subtree
349             subtree_root.left_index = left_child.right_index;
350             //original root adopted a subtree (whose height did not change), so
                    update its height
351             subtree_root.update_height(nodes);
352
353             //left child adopts original root and its children
354             left_child.right_index = subtree_root_index;
355             //left child (new root) adopted the original root (whose height has been
                    updated), so update its height
356             left_child.update_height(nodes);
357             //since left child took the subtree root's place, it has the same number
                    of children as the original root
358             left_child.num_children = subtree_root.num_children;
359
360             //root has new children, so update that counter (done after changing the
                    left child's children counter
361             //because that depends on the original root's counter)
362             subtree_root.num_children = 0;
363             if (subtree_root.left_index != 0)
364                 subtree_root.num_children += 1 +
                        nodes[subtree_root.left_index].num_children;
365             if (subtree_root.right_index != 0)
366                 subtree_root.num_children += 1 +
                        nodes[subtree_root.right_index].num_children;
367
368             //set the left child as the new root
369             subtree_root_index = left_child_index;
370         }
```

```
371         int do_search(size_t nodes_visited, //starts at 0 when this function is
                first called (ie does not include current node visitation)
372                 size_t subtree_root_index,
373                 Key const& key,
374                 Value& value,
375                 bool& found_key) const
376         {
377             if (subtree_root_index != 0) {
378                 Node const& subtree_root = nodes[subtree_root_index];
379                 ++nodes_visited;
380                 if (key < subtree_root.key) {
381                     nodes_visited = do_search(nodes_visited, subtree_root.left_index,
                            key, value, found_key);
382                 } else if (key > subtree_root.key) {
383                     nodes_visited = do_search(nodes_visited,
                            subtree_root.right_index, key, value, found_key);
384                 } else if (key == subtree_root.key) {
385                     //found key, replace the value
386                     value = subtree_root.value;
387                     found_key = true;
388                 } else {
389                     throw std::logic_error("Unexpected compare result");
390                 }
391             }
392             return nodes_visited;
393         }
394         void prepare_cluster_distribution(size_t subtree_root_index,
395                                           size_t curr_height, //includes the height of
                                                the current node, ie assumes current node
                                                exists
396                                           size_t cluster_counter[])
397         {
398             Node const& subtree_root = nodes[subtree_root_index];
399             if ( ! subtree_root.left_index && ! subtree_root.right_index)
400                 //at a leaf node
401                 cluster_counter[curr_height]++;
402             else {
403                 if (subtree_root.left_index)
404                     prepare_cluster_distribution(subtree_root.left_index, curr_height
                            + 1, cluster_counter);
405                 if (subtree_root.right_index)
406                     prepare_cluster_distribution(subtree_root.right_index,
                            curr_height + 1, cluster_counter);
407             }
408         }
409
410         void remove_ith_node_inorder(size_t& subtree_root_index,
411                                      size_t& ith_node_to_delete,
412                                      Key& key)
413         {
414             Node& subtree_root = nodes[subtree_root_index];
415             if (subtree_root.left_index)
```

```
416                 remove_ith_node_inorder(subtree_root.left_index, ith_node_to_delete,
                        key);
417             if (ith_node_to_delete == 0)
418                 //deleted node in child subtree; nothing more to do
419                 return;
420             if (--ith_node_to_delete == 0) {
421                 //delete the current node
422                 value_type dummy_val;
423                 remove(subtree_root.key.raw_copy(), dummy_val);
424                 key = subtree_root.key;
425                 return;
426             }
427             if (subtree_root.right_index)
428                 remove_ith_node_inorder(subtree_root.right_index,
                        ith_node_to_delete, key);
429         }
430
431     public:
432         /*
433             The constructor will allocate an array of capacity (binary
434             tree) nodes. Then make a chain from all the nodes (e.g.,
435             make node 2 the left child of node 1, make node 3 the left
436             child of node 2, &c. this is the initial free list.
437         */
438         BST(size_t capacity):
439             curr_capacity(capacity)
440         {
441             if (capacity == 0) {
442                 throw std::domain_error("capacity must be at least 1");
443             }
444             nodes = new Node[capacity + 1];
445             clear();
446         }
447         /*
448             if there is space available, adds the specified key/value-pair to the
                    tree
449             and returns the number of nodes visited, V; otherwise returns -1 * V. If
                    an
450             item already exists in the tree with the same key, replace its value.
451         */
452         virtual int insert(key_type const& key, value_type const& value) {
453             if (size() == capacity())
454                 //no more space
455                 return -1 * size();
456             bool found_key = false;
457             Key k(key);
458             Value v(value);
459             int nodes_visited = insert_at_leaf(0, root_index, k, v, found_key);
460             if (_DEBUG_)
461                 this->nodes[this->root_index].validate_children_count_recursive(this->nodes);
462             return nodes_visited;
463         }
```

```
464         /*
465             if there is an item matching key, removes the key/value-pair from the
                tree, stores
466             it's value in value, and returns the number of probes required, V;
                otherwise returns -1 * V.
467         */
468         virtual int remove(key_type const& key, value_type& value) {
469             if (is_empty())
470                 return 0;
471             bool found_key = false;
472             Key k(key);
473             Value v(value);
474             int nodes_visited = do_remove(0, root_index, k, v, found_key);
475             if (_DEBUG_)
476                 this->nodes[this->root_index].validate_children_count_recursive(this->nodes);
477             if (found_key)
478                 value = v.raw_copy();
479             return found_key ? nodes_visited : -1 * nodes_visited;
480         }
481         /*
482             if there is an item matching key, stores it's value in value, and
                returns the number
483             of nodes visited, V; otherwise returns -1 * V. Regardless, the item
                remains in the tree.
484         */
485         virtual int search(key_type const& key, value_type& value) {
486             if (is_empty())
487                 return 0;
488             bool found_key = false;
489             Key k(key);
490             Value v(value);
491             int nodes_visited = do_search(0, root_index, k, v, found_key);
492             if (found_key)
493                 value = v.raw_copy();
494             return found_key ? nodes_visited : -1 * nodes_visited;
495         }
496         /*
497             removes all items from the map
498         */
499         virtual void clear() {
500             //Since I use size_t to hold the node indices, I make the node array
501             //1-based, with child index of 0 indicating that the current node is a
                leaf
502             for (size_t i = 1; i != capacity(); ++i)
503                 nodes[i].disable_and_adopt_free_tree(i + 1);
504             free_index = 1;
505             root_index = 0;
506         }
507         /*
508             returns true IFF the map contains no elements.
509         */
510         virtual bool is_empty() const {
```

```
511             return size() == 0;
512         }
513         /*
514             returns the number of slots in the backing array.
515         */
516         virtual size_t capacity() const {
517             return curr_capacity;
518         }
519         /*
520             returns the number of items actually stored in the tree.
521         */
522         virtual size_t size() const {
523             if (root_index == 0) return 0;
524             Node const& root = nodes[root_index];
525             return 1 + root.num_children;
526         }
527         /*
528             [not a regular BST operation, but specific to this implementation]
529             returns the tree's load factor: load = size / capacity.
530         */
531         virtual double load() const {
532             return static_cast<double>(size()) / capacity();
533         }
534         /*
535             prints the tree in the following format:
536             +--[tiger]
537             |  |
538             |  |  +--[panther]
539             |  |  |
540             |  +--[ocelot]
541             |     |
542             |     +--[lion]
543             |
544             [leopard]
545             |
546             |     +--[house cat]
547             |     |
548             |  +--[cougar]
549             |  |
550             +--[cheetah]
551                |
552                +--[bobcat]
553         */
554         virtual std::ostream& print(std::ostream& out) const {
555             if (is_empty())
556                 return out;
557             size_t num_lines = size() * 2 - 1;
558             //use CDAL here so we can print really super-huge trees where the write
                    buffer doesn't fit in memory
559             CDAL<std::string> buffer_lines(100000);
560             for(size_t i = 0; i <= num_lines; ++i)
561                 buffer_lines.push_back("");
```

```
562             Node const& root = nodes[root_index];
563             size_t root_line_index = 1;
564             if (root.right_index) {
565                 root_line_index += 2 * (1 + nodes[root.right_index].num_children);
566             }
567             write_subtree_buffer(root_index, buffer_lines, root_line_index, 1,
                    num_lines + 1);
568             for (size_t i = 1; i <= num_lines; ++i)
569                 out << buffer_lines[i] << std::endl;
570             return out;
571         }
572
573         /*
574             returns a list indicating the number of leaf nodes at each height (since
                    the RBST doesn't exhibit
575             true clustering, but can have degenerate branches).
576         */
577         virtual priority_queue<hash_utils::ClusterInventory> cluster_distribution()
                {
578             //use an array to count cluster instances, then feed those to a priority
                    queue and return it.
579             priority_queue<ClusterInventory> cluster_pq;
580             if (is_empty()) return cluster_pq;
581             size_t max_height = nodes[root_index].height;
582             size_t cluster_counter[max_height + 1];
583             for (size_t i = 0; i <= max_height; ++i)
584                 cluster_counter[i] = 0;
585             prepare_cluster_distribution(root_index, 1, cluster_counter);
586             for (size_t i = 1; i <= max_height; ++i)
587                 if (cluster_counter[i] > 0) {
588                     ClusterInventory cluster{i, cluster_counter[i]};
589                     cluster_pq.add_to_queue(cluster);
590                 }
591             return cluster_pq;
592         }
593
594         /*
595             generate a random number, R, (1,size), and starting with the root (node
                    1), do an in-order
596             traversal to find the R-th occupied node; remove that node (adjusting
                    its children accordingly),
597             and return its key.
598
599             ***XXX: this likely contains a bug when using const char* keys in that
                    we'll be returning a dangling pointer!!!***
600         */
601         virtual key_type remove_random() {
602             if (size() == 0) throw std::logic_error("Cant remove from an empty map");
603             size_t ith_node_to_delete = 1 + rand_i(size());
604             Key key;
605             remove_ith_node_inorder(root_index, ith_node_to_delete, key);
606             key_type ret = key.raw_copy();
```

```
607             return ret;
608         }
609     };
610 }
611
612 #endif
```

part4/source/rbst.h

part4/source/rbst.h

```
#ifndef _RBST_H_
#define _RBST_H_


#include <cstdlib>
#include <sstream>
#include "../../common/CDAL.h"
#include "../../common/common.h"
#include "../../common/priority_queue.h"
#include "bst.h"

namespace cop3530 {
    template<typename key_type,
             typename value_type>
    class RBST: public BST<key_type, value_type> {
    /*
        Within the RBST insert_at_leaf method, the recursive execution path is randomly redirected
        to insert at the root. Therefore, we simply inherit from a generic BST class and wrap the
        insert_at_leaf method with that potential alternative execution path
    */
    private:
        using super = BST<key_type, value_type>;
        using typename super::Node;
        typedef hash_utils::Key<key_type> Key;
        typedef hash_utils::Value<value_type> Value;
        int insert_at_leaf(size_t nodes_visited, //starts at 0 when this function is first called (ie does not include current node visitation)
                           size_t& subtree_root_index,
                           Key const& key,
                           Value const& value,
                           bool& found_key)
        {
            //parent was not a leaf
            Node& subtree_root = this->nodes[subtree_root_index];
            if (rand() < RAND_MAX / (subtree_root.num_children + 1)) {
                //randomly insert at the subtree root
                nodes_visited = insert_at_root(nodes_visited, subtree_root_index, key, value, found_key);
            } else {
                nodes_visited = super::insert_at_leaf(nodes_visited, subtree_root_index, key, value, found_key);
            }
            return nodes_visited;
        }
        int insert_at_root(size_t nodes_visited,
```

```
43                      size_t& subtree_root_index,
44                      Key const& key,
45                      Value const& value,
46                      bool& found_key)
47          {
48              if (subtree_root_index == 0) {
49                  //parent was a leaf, so create a new leaf
50                  subtree_root_index = this->procure_node(key, value);
51              } else {
52                  //parent was not a leaf
53                  Node& subtree_root = this->nodes[subtree_root_index];
54                  ++nodes_visited;
55                  //keep going down to the base of the tree
56
57                  if (key < subtree_root.key) {
58                      nodes_visited = insert_at_root(nodes_visited,
                            subtree_root.left_index, key, value, found_key);
59                      if ( ! found_key) {
60                          //new node currently a new child of subtree root, so increment
61                          //the subtree root's number of children before rotating - new
                                node
62                          //will adopt the root and its children and will take on the
                                value
63                          //of its num_children
64                          subtree_root.num_children++;
65
66                          //current subtree root may have had its height changed, so
                                update that before
67                          //promoting the new node
68                          subtree_root.update_height(this->nodes);
69                          this->rotate_right(subtree_root_index);
70                      }
71                  } else if (key > subtree_root.key) {
72                      nodes_visited = insert_at_root(nodes_visited,
                            subtree_root.right_index, key, value, found_key);
73                      if ( ! found_key) {
74                          subtree_root.num_children++;
75
76                          //current subtree root may have had its height changed, so
                                update that before
77                          //promoting the new node
78                          subtree_root.update_height(this->nodes);
79                          this->rotate_left(subtree_root_index);
80                      }
81                  } else if (key == subtree_root.key) {
82                      //found key, replace the value
83                      subtree_root.value = value;
84                      found_key = true;
85                  } else {
86                      throw std::logic_error("Unexpected compare result");
87                  }
88              }
```

```
89              return nodes_visited;
90          }
91      public:
92          RBST(size_t capacity): super(capacity) {}
93          /*
94              if there is space available, adds the specified key/value-pair to the
                    tree
95              and returns the number of nodes visited, V; otherwise returns -1 * V. If
                    an
96              item already exists in the tree with the same key, replace its value.
97          */
98          int insert(key_type const& key, value_type const& value) {
99              if (this->size() == this->capacity())
100                 //no more space
101                 return -1 * this->size();
102             bool found_key = false;
103             Key k(key);
104             Value v(value);
105             int nodes_visited = insert_at_leaf(0, this->root_index, k, v, found_key);
106             if (_DEBUG_)
107                 this->nodes[this->root_index].validate_children_count_recursive(this->nodes);
108             return nodes_visited;
109         }
110     };
111 }
112
113 #endif
```

Part IV BONUS: AVL Tree

Part 4 AVL Testing Strategy

Paul Nickerson

operation_failures.cpp

Within the failure operations scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start with an empty map and try to search() and remove() an item whose key does not exist in the map. Both calls should return a value less than zero (indicating key not found). I then fill the map with a bunch of items (it is impossible to run out of space because collisions are resolved via an arbitrarily-growable liked list). From this newly-filled map, I attempt to remove() a key that doesn't exist in the map, and search() for a key that does not exist in the map, both of which should return a value less than zero.

The AVL class keeps track of the height and number of children of each subtree with O(1) complexity during each operation that potentially changes those values. Because these values are so crucial to the map's functionality, and considering the difficulty of externally validating those values without adding more public methods, I include a preprocessor macro, _DEBUG_, which, when set to true (it defaults to false in production uses of the class), indicates to the BST base class to recursively verify the value of these values before each public method returns. This is potentially an expensive O(N) operation that significantly slows down map functionality, which is why it is disabled by default and specifically enabled during unit testing. If it is determined that the O(1) calculated values do not match the O(N), "true" values, the class throws an exception, which is caught by the CATCH testing framework and causes the test to fail.

In addition, the AVL map maintains a balance factor between -1 and 1 at every subtree. I included another recursive function, enabled when _DEBUG_==true, to check for any subtrees that have abs(balance factor) > 1. If it finds any, it throws an exception which is caught by the CATCH testing framework.

operation_successes.cpp

Within the success operations scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start by filling the map with a bunch of items, clearing it, then filling it up again. The map should then report the correct size. I check that several keys which are expected to exist in the map actually do exist (including the lowest possible key, the highest possible key, and one in the middle).

I check the print() function by routing it to an output string stream and count the number of hyphens in the output, which indicate empty slots. Since load factor = occupied buckets /

capacity, we can get the number of unoccupied buckets as capacity * (1 - load). This should equal the number of hyphens in the print() output.

I then attempt to remove() several keys which are known to exist, and check that their associated values are what were expected. After these items are removed, I try to both search() and remove() them, which should all return false.

I test remove_random() in a similar way to the preceeding remove() check. I remove a random key, then try to search() for it and remove() it. Both checks should fail and return a value less than zero.

tree_structure.cpp

Because I was able to successfully implement the pretty-print bonus method, it is fairly straightforward to verify that the underlying AVL map successfully maintains the correct BST tree structure. I print the tree structure to a file, then make a system() call to pipe that through a few command line utilities that extract just the numerical key (I say numerical, but that number could be in the form of a string of const char*). This results in the natural order of the keys as they exist in the tree. I pipe those through uniq and sort to remove duplicates and induce expected sorting order. If the tree exhibits the correct structure, the output of those two operations will match verbatim.

I start by filling the map to half capacity, verify that load() returns 0.5, then check validate the tree structure. Afterwards, I fill the tree to full capacity and then delete half the nodes via remove() and remove_random() operations in an attempt to destablize tree structure. Then I validate the tree structure again.

This series of checks is implemented for each of the four supported key types.

part4_bonus/part4bonus.pdf

part4_bonus/checklist.txt

```
AVL Tree written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part IV BONUS: AVL Tree
=============================================================================
My MAP implementation uses the data structure described in the part IV
instructions and conforms to the technique required for this map
variety: yes

My MAP implementation 100% correctly implements AVL tree behavior: yes

My MAP implementation 100% correctly supports the following key types:
* signed int: yes
* double: yes
* c-string: yes
* std::string: yes

My MAP implementation 100% correctly supports the ANY value type: yes

My MAP implementation 100% correctly supports the following methods
as described in part IV:

* insert: yes
* remove: yes
* search: yes
* search: yes
* clear: yes
* is_empty: yes
* capacity: yes
* size: yes
* load: yes
* print: yes
* cluster_distribution(): yes
* remove_random(): yes

My MAP implementation 100% correctly implements the bonus print(): yes


=============================================================================
FOR ALL PARTS
=============================================================================

My MAP implementation compiles correctly using g++ v4.8.2 on the
```

```
OpenBSD VM: yes

My TEST compiles correctly using g++ v4.8.2 on the OpenBSD VM: yes

My TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this AVL Tree
and the associated tests.
Paul Nickerson, Dec 2 in COP3520 section 1087

------------------------------------------------------------------------
------------------------------------------------------------------------

How to compile and run my unit tests on the OpenBSD VM
cd part4bonus/source
./compile.sh
./run_tests > output.txt
```

common/common.h

common/common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#ifndef _DEBUG_
    //various internal integrity checks that can be expensive, we want to disable them in production
    #define _DEBUG_ false
#endif

#include <string.h>
#include <limits>
#include <stdexcept>
#include <ostream>
#include <cmath>

namespace cop3530 {
    inline double lg(size_t i) {
        return std::log(i) / std::log(2);
    }
    inline size_t rand_i(size_t max) {
        size_t bucket_size = RAND_MAX / max;
        size_t num_buckets = RAND_MAX / bucket_size;
        size_t big_rand;
        do {
               big_rand = rand();
        } while(big_rand >= num_buckets * bucket_size);
        return big_rand / bucket_size;
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        static constexpr size_t primes[] = { //from algorithms in c++, helps us to choose a prime-number map capacity
            251,
            509,
            1021,
            2039,
            4093,
            8191,
            16381,
            32749,
            65521,
            131071,
            262193,
            524287,
            1048573,
            2097143,
```

```
            4194301,
            8388593,
            16777213,
            33554393,
            67108859,
            134217689,
            268435399,
            536870909,
            1073741789,
            2147483647
        };
        struct ClusterInventory {
            size_t cluster_size;
            size_t num_instances;
            struct cluster_size_less_predicate {
                bool operator()(ClusterInventory const& cluster1, ClusterInventory
                     const& cluster2) {
                    return cluster1.cluster_size < cluster2.cluster_size;
                }
            };
        };

        inline size_t str_to_numeric(const char* str) {
            unsigned int base = 257; //prime number chosen near an 8-bit character
            size_t numeric = 0;
            for (; *str != 0; ++str)
                numeric = numeric * base + *str;
            return numeric;
        }
        namespace functors {

            struct map_capacity_planner {
                size_t operator()(size_t min_capacity) {
                    for (int i = 0; i != 24; ++i)
                        if (min_capacity < primes[i])
                            return primes[i];
                    throw std::domain_error("Provided min capacity too large.
                         Consider extending the list of prime numbers");
                }
            };
            struct compare {
                int operator()(const char* a, const char* b) const {
                    int cmp = strcmp(a, b);
                    return (cmp < 0 ? -1 :
                                    (cmp > 0 ? 1 : 0));
                }
                int operator()(double a, double b) const {
                    return (a < b ? -1 :
                                  (a > b ? 1 : 0));
                }
                int operator()(std::string const& a, std::string const& b) const {
                    return (a < b ? -1 :
```

```
                        (a > b ? 1 : 0));
    }
    int operator()(int a, int b) const {
        return (a < b ? -1 :
                        (a > b ? 1 : 0));
    }
};
namespace primary_hashes {
    struct hash_basic {
    //this is such a stupid hash method, but unlike my pathetic attempts at implementing
    //various other hashing methods, it works and is generalizable to all the required key
    //types. together with double hashing it should make for a passable hashing routine.
    public:
        size_t operator()(const char* key) const {
            return str_to_numeric(key);
        }
        size_t operator()(double key) const {
            return static_cast<size_t>(std::fmod(key, max_size_t));
        }
        size_t operator()(int key) const {
            return static_cast<size_t>(key);
        }
        size_t operator()(std::string const& key) const {
            const char* c_key = key.c_str();
            return operator()(c_key);
        }
    };
}
namespace secondary_hashes {
    struct linear_probe {
        bool changes_with_probe_attempt() const {
            return false;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return 1;
        }
    };
    struct quadratic_probe {
        bool changes_with_probe_attempt() const {
            return true;
        }
        template<typename T>
        size_t operator()(T unused, size_t probe_attempt) const {
            return probe_attempt;
        }
    };
    struct hash_double {
    private:
```

```
145                 size_t hash_numeric(size_t numeric) const {
146                     size_t hash = numeric % 97; //simple modulus using a prime
                            number (from algorithms in c++)
147                     //the second hash may not be zero (will cause an infinite
                            loop).
148                     //also, hash must be relatively prime to map_capacity so that
                            every slot can be hit.
149                     //map capacity is a prime number based chosen from the table,
                            so any value less than
150                     //map capacity should work
151                     return hash;
152                 }
153             public:
154                 bool changes_with_probe_attempt() const {
155                     return false;
156                 }
157                 size_t operator()(const char* key, size_t) const {
158                     size_t numeric = str_to_numeric(key);
159                     return hash_numeric(numeric);
160                 }
161                 size_t operator()(double key, size_t) const {
162                     return hash_numeric(key);
163                 }
164                 size_t operator()(int key, size_t) const {
165                     return hash_numeric(key);
166                 }
167                 size_t operator()(std::string key, size_t) const {
168                     const char* c_key = key.c_str();
169                     return operator()(c_key, 0);
170                 }
171             };
172         }
173     }
174
175     template<typename T>
176     class GenericContainer {
177         /*
178             for the types we need to support other than const char* (ie int,
                    double, and std::string),
179             we can pass these around willy-nilly. for const char*, handled
                    below, we will obtain our
180             own copy of the character array by wrapping it in a std::string
181         */
182     private:
183         T raw;
184         functors::compare compare;
185     public:
186         GenericContainer(const T& val): raw(val) {}
187         GenericContainer() = default;
188         GenericContainer& operator=(GenericContainer const& rhs) = delete;
189         T operator()() const {
190             return raw;
```

```
191             }
192             T copy() const {
193                 return raw;
194             }
195             void reset(const T& val) {
196                 raw = val;
197             }
198             int compare_to(GenericContainer const& other) const {
199                 return compare(raw, other.raw);
200             }
201         };
202         template<>
203         class GenericContainer<const char*> {
204             /*
205                 class template specialization for character arrays, stores a local
                        copy of the character array
206             */
207         private:
208             char* raw = nullptr;
209             functors::compare compare;
210         public:
211             GenericContainer(const char* val) {
212                 reset(val);
213             }
214             GenericContainer() = default;
215             const char* operator()() const {
216                 return raw;
217             }
218             const char* copy() const {
219                 if (raw == nullptr) return nullptr;
220                 size_t len = strlen(raw);
221                 char* new_str = new char[len + 1];
222                 strncpy(new_str, raw, len);
223                 new_str[len] = 0;
224                 return new_str;
225             }
226             void reset(const char* val) {
227                 if (raw) {
228                     delete raw;
229                     raw = nullptr;
230                 }
231                 if (val != nullptr) {
232                     size_t len = strlen(val);
233                     raw = new char[len + 1];
234                     strncpy(raw, val, len);
235                     raw[len] = 0;
236                 }
237             }
238             int compare_to(GenericContainer const& other) const {
239                 return compare(raw, other.raw);
240             }
241         };
```

```
242
243         template<typename key_type,
244                  typename primary_hash =
                             hash_utils::functors::primary_hashes::hash_basic,
245                  typename secondary_hash =
                             hash_utils::functors::secondary_hashes::hash_double>
246         class Key {
247         private:
248             GenericContainer<key_type> raw_key;
249             primary_hash hasher1;
250             secondary_hash hasher2;
251             size_t hash1_val;
252             size_t hash2_val;
253         public:
254             Key& operator=(Key const& rhs) {
255                 if (&rhs == this)
256                     return *this;
257                 reset(rhs.raw_key());
258             }
259             bool operator==(Key const& rhs) const {
260                 return raw_key.compare_to(rhs.raw_key) == 0;
261             }
262             bool operator<(Key const& rhs) const {
263                 return raw_key.compare_to(rhs.raw_key) == -1;
264             }
265             bool operator>(Key const& rhs) const {
266                 return raw_key.compare_to(rhs.raw_key) == 1;
267             }
268             bool operator!=(Key const& rhs) const {
269                 return ! operator==(rhs);
270             }
271             size_t hash(size_t map_capacity, size_t probe_attempt) const {
272                 size_t local_hash2_val;
273                 if (probe_attempt != 0 && hasher2.changes_with_probe_attempt())
274                 {
275                     //if the hashing function value is dependent on the probe attempt
276                     //(eg quadratic probing), then we need to retrieve the new value*/
277                     local_hash2_val = hasher2(raw_key(), probe_attempt);
278                 } else {
279                     //otherwise we can just use the value we have stored
280                     local_hash2_val = hash2_val;
281                 }
282                 return (hash1_val + probe_attempt * local_hash2_val) % map_capacity;
283             }
284             key_type raw() const {
285                 return raw_key();
286             }
287             key_type raw_copy() const {
288                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
289                 return raw_key.copy();
290             }
```

```
291             template<typename T>
292             void reset(T key) {
293                 raw_key.reset(key);
294                 size_t base_probe_attempt = 0;
295                 hash1_val = hasher1(key);
296                 hash2_val = hasher2(key, base_probe_attempt);
297             }
298             void reset(const char* key) {
299                 raw_key.reset(key);
300                 if (key != nullptr) {
301                     size_t base_probe_attempt = 0;
302                     hash1_val = hasher1(key);
303                     hash2_val = hasher2(key, base_probe_attempt);
304                 }
305             }
306             explicit Key(key_type const& key): raw_key(key) {
307                 reset(key);
308             }
309             Key() = default;
310         };
311         template <typename value_type>
312         class Value {
313         private:
314             functors::compare compare;
315             GenericContainer<value_type> raw_value;
316         public:
317             Value& operator=(Value const& rhs) {
318                 if (&rhs == this)
319                     return *this;
320                 reset(rhs.raw_value());
321             }
322             bool operator==(Value const& rhs) const {
323                 return compare(raw_value(), rhs.raw_value());
324             }
325             bool operator==(value_type const& rhs) const {
326                 return compare(raw_value(), rhs) == 0;
327             }
328             value_type raw() const {
329                 return raw_value();
330             }
331             value_type raw_copy() const {
332                 //this is what is returned to the client, who is responsible for
                        deleting it if its, eg a pointer to a character array
333                 return raw_value.copy();
334             }
335             void reset(value_type value) {
336                 raw_value.reset(value);
337             }
338             explicit Value(value_type const& value): raw_value(value) {}
339             Value() = default;
340         };
341     }
```

```
342 }
343
344 #endif
```

priority_queue.h

priority_queue.h

```
#ifndef _PRIORITY_QUEUE_H_
#define _PRIORITY_QUEUE_H_

#include "SDAL.h"
#include "common.h"

namespace cop3530 {
    //this class allows efficient sorting clusters by size for the
        cluster_distribution functions
    template<typename T,
            typename PriorityCompare =
                cop3530::hash_utils::ClusterInventory::cluster_size_less_predicate>
    class priority_queue {
    private:
        PriorityCompare first_arg_higher_priority;
        //SDAL has all the benefits of std::vector (ie fast random access and
            automatic resizing)
        //while having the added benefit of being legal to use in cop3530
        SDAL<T> tree;
        size_t num_items = 0;
        void fix_up(size_t index) {
            while (index > 1
                  && first_arg_higher_priority(tree[index], tree[index / 2]))
            {
                std::swap(tree[index / 2], tree[index]);
                index /= 2;
            }
        }
        void fix_down() {
            size_t parent_index = 1;
            while (2 * parent_index <= num_items) {
                size_t left_index = 2 * parent_index;
                size_t right_index = left_index + 1;
                size_t higher_priority_index = left_index;
                if (right_index <= num_items
                    && first_arg_higher_priority(tree[right_index], tree[left_index]))
                {
                    higher_priority_index = right_index;
                }
                if ( ! first_arg_higher_priority(tree[higher_priority_index],
                     tree[parent_index]))
                    //no more items to elevate
                    break;
                std::swap(tree[parent_index], tree[higher_priority_index]);
                parent_index = higher_priority_index;
            }
        }
```

```
44      public:
45          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
46              T empty_item;
47              tree.push_back(empty_item);
48          }
49          priority_queue(priority_queue const& src) {
50              tree = src.tree;
51              num_items = src.num_items;
52          }
53          T get_next_item() {
54              std::swap(tree[1], tree[num_items]);
55              T ret = tree[num_items--];
56              fix_down();
57              return ret;
58          }
59          void add_to_queue(T const& item) {
60              tree.push_back(item);
61              num_items++;
62              fix_up(num_items);
63          }
64          size_t size() {
65              return num_items;
66          }
67          bool empty() {
68              return num_items == 0;
69          }
70      };
71  }
72
73  #endif // _PRIORITY_QUEUE_H_
```

common/unit_test_utils.h

common/unit_test_utils.h

```
#ifndef _UNIT_TEST_UTILS_H_
#define _UNIT_TEST_UTILS_H_

#include <iostream>
#include <string>
#include <fstream>

namespace cop3530 {
    namespace unit_test_utils {
        inline std::string guid() {
            std::string ret = "";
            for (size_t i = 0; i != 32; ++i) {
                size_t rnd = cop3530::rand_i(16);
                if (rnd < 10)
                    ret += std::string(1, '0' + rnd);
                else
                    ret += std::string(1, 'A' + rnd - 10);
            }
            return ret;
        }
        inline std::string get_tmp_filename() {
            return std::string("/tmp/") + guid() + std::string(".out");
        }
        inline void delete_file(std::string file_path) {
            system((std::string("rm ") + file_path + std::string(" 2>&1 >>
                /tmp/debug")).c_str());
        }
        inline std::string shell_cmd(std::string cmd) {
            std::string shell_script_file = get_tmp_filename();
            std::string output_file = get_tmp_filename();
            std::ofstream shell_script_out(shell_script_file);
            shell_script_out << "#!/bin/sh" << std::endl << cmd << std::endl;
            shell_script_out.close();
            std::string chmod_cmd = std::string("chmod +x ") + shell_script_file;
            system(chmod_cmd.c_str());
            std::string invoke_cmd = shell_script_file + std::string(" > ") +
                output_file;
            system(invoke_cmd.c_str());
            std::ifstream read_output(output_file);
            std::ostringstream oss;
            std::string tmp;
            while (std::getline(read_output, tmp)) {
                oss << tmp;
                if ( ! read_output.eof())
                    oss << "\n";
            }
            delete_file(output_file);
```

```
            delete_file(shell_script_file);
            return oss.str();
        }
    }
}

#endif // _UNIT_TEST_UTILS_H_
```

part4_bonus/source/bst.h

part4_bonus/source/bst.h

```
#ifndef _BST_H_
#define _BST_H_

#include <cstdlib>
#include <sstream>
#include "../../common/CDAL.h"
#include "../../common/common.h"
#include "../../common/priority_queue.h"

namespace cop3530 {
    template<typename key_type,
             typename value_type>
    class BST {
    protected: //let RBST and AVL inherit everything
        typedef hash_utils::ClusterInventory ClusterInventory;
        typedef hash_utils::Key<key_type> Key;
        typedef hash_utils::Value<value_type> Value;
        struct Node;
        typedef Node* link;
        struct Node {
            Key key;
            Value value;
            size_t num_children;
            size_t left_index;
            size_t right_index;
            size_t height; //height tracking coded in this class, but not used (for
                AVL, which is this class with self-balancing)
            bool is_occupied;
            size_t validate_children_count_recursive(Node* nodes) {
                //this function is for debugging purposes, does recursive traversal
                    to find the correct number of children
                size_t child_count = 0;
                if (left_index)
                    child_count += 1 +
                        nodes[left_index].validate_children_count_recursive(nodes);
                if (right_index)
                    child_count += 1 +
                        nodes[right_index].validate_children_count_recursive(nodes);
                if (child_count != num_children) {
                    std::ostringstream msg;
                    msg << "Manually counted children, " << child_count << ",
                        different than child count, " << num_children;
                    throw std::logic_error(msg.str());
                }
                return child_count;
            }
            size_t get_height_recursive(Node* nodes) {
```

```
43              //this function is for debugging purposes, does recursive traversal
                    to find the correct height
44              size_t left_height = 0, right_height = 0;
45              size_t calculated_height = 0;
46              if (left_index)
47                  left_height = nodes[left_index].get_height_recursive(nodes);
48              if (right_index)
49                  right_height = nodes[right_index].get_height_recursive(nodes);
50              calculated_height = 1 + std::max(left_height, right_height);
51              return calculated_height;
52          }
53          void update_height(Node* nodes) {
54              //note: this method depends on the left and right subtree heights
                    being correct
55              size_t left_height = 0, right_height = 0;
56              if (left_index)
57                  left_height = nodes[left_index].height;
58              if (right_index)
59                  right_height = nodes[right_index].height;
60              height = 1 + std::max(left_height, right_height);
61              if (_DEBUG_) {
62                  size_t calculated_height = get_height_recursive(nodes);
63                  if (calculated_height != height) {
64                      std::ostringstream msg;
65                      msg << "Manually calculated height, " << calculated_height <<
                            ", different than tracked height, " << height;
66                      throw std::logic_error(msg.str());
67                  }
68              }
69          }
70          void disable_and_adopt_free_tree(size_t free_index) {
71              is_occupied = false;
72              height = 0;
73              num_children = 0;
74              right_index = 0;
75              left_index = free_index;
76          }
77          void reset_and_enable(Key const& new_key, Value const& new_value) {
78              is_occupied = true;
79              height = 1; //self
80              left_index = right_index = 0;
81              num_children = 0;
82              key = new_key;
83              value = new_value;
84          }
85          int balance_factor(const Node* nodes) const {
86              size_t left_height = 0, right_height = 0;
87              if (left_index)
88                  left_height = nodes[left_index].height;
89              if (right_index)
90                  right_height = nodes[right_index].height;
```

```
91              return static_cast<long int>(left_height) - static_cast<long
                    int>(right_height);
92          }
93      };
94      Node* nodes; //***note: array is 1-based so leaf nodes have child indices
            set to zero
95      size_t free_index;
96      size_t root_index;
97      size_t curr_capacity;
98      virtual size_t remove_smallest_key_node_index(size_t& subtree_root_index) {
99          //returns the index of the node with the smallest key, while
100         //setting its parent's left child index to the smallest key node's
101         //right child index. recursion downward through this function updates
102         //the heights of the nodes it traverses
103         Node& subtree_root = nodes[subtree_root_index];
104         size_t smallest_key_node_index = 0;
105         if (subtree_root_index == 0) {
106             throw std::logic_error("Expected to find a valid node, but didn't");
107         } else {
108             if (subtree_root.left_index) {
109                 smallest_key_node_index =
                        remove_smallest_key_node_index(subtree_root.left_index);
110                 subtree_root.num_children--;
111                 subtree_root.update_height(nodes);
112             } else {
113                 smallest_key_node_index = subtree_root_index;
114                 subtree_root_index = subtree_root.right_index;
115             }
116         }
117         return smallest_key_node_index;
118     }
119     virtual size_t remove_largest_key_node_index(size_t& subtree_root_index) {
120         //returns the index of the node with the largest key, while
121         //setting its parent's right child index to the largest key node's
122         //left child index. recursion downward through this function updates
123         //the heights of the nodes it traverses
124         Node& subtree_root = nodes[subtree_root_index];
125         size_t largest_key_node_index = 0;
126         if (subtree_root_index == 0) {
127             throw std::logic_error("Expected to find a valid node, but didn't");
128         } else {
129             if (subtree_root.right_index) {
130                 largest_key_node_index =
                        remove_largest_key_node_index(subtree_root.right_index);
131                 subtree_root.num_children--;
132                 subtree_root.update_height(nodes);
133             } else {
134                 largest_key_node_index = subtree_root_index;
135                 subtree_root_index = subtree_root.left_index;
136             }
137         }
138         return largest_key_node_index;
```

```
139         }
140         virtual void remove_node(size_t& subtree_root_index) {
141             Node& subtree_root = nodes[subtree_root_index];
142             size_t index_to_delete = subtree_root_index;
143             if (subtree_root.right_index || subtree_root.left_index) {
144                 //subtree has at least one child
145                 if (subtree_root.right_index)
146                     //replace the root with the smallest-keyed node in the right
                            subtree
147                     subtree_root_index =
                            remove_smallest_key_node_index(subtree_root.right_index);
148                 else if (subtree_root.left_index)
149                     //replace the root with the largest-keyed node in the left subtree
150                     subtree_root_index =
                            remove_largest_key_node_index(subtree_root.left_index);
151                 //have the new root adopt the old root's children
152                 Node& new_root = nodes[subtree_root_index];
153                 new_root.left_index = subtree_root.left_index;
154                 new_root.right_index = subtree_root.right_index;
155                 //the new root has the same number of children as the old root,
                        minus one
156                 new_root.num_children = subtree_root.num_children - 1;
157                 //removing the smallest/largest-keyed node from the old root has the
                        effect of
158                 //updating the heights of the old root's relevant subtrees (which
                        the new root
159                 //just adopted), so we can update the new root's height now
160                 new_root.update_height(nodes);
161             } else
162                 //neither subtree exists, so just delete the node
163                 subtree_root_index = 0;
164             //node has been disowned by all ancestors, and has disowned all
                    descendents, so free it
165             add_node_to_free_tree(index_to_delete);
166         }
167         virtual int do_remove(size_t nodes_visited, //starts at 0 when this
                function is first called (ie does not include current node visitation)
168                               size_t& subtree_root_index,
169                               Key const& key,
170                               Value& value,
171                               bool& found_key)
172         {
173             if (subtree_root_index != 0) {
174                 Node& subtree_root = nodes[subtree_root_index];
175                 ++nodes_visited;
176                 //keep going down to the base of the tree
177                 if (key < subtree_root.key) {
178                     nodes_visited = do_remove(nodes_visited, subtree_root.left_index,
                            key, value, found_key);
179                     if (found_key) {
180                         //found the desired node and delete it
181                         subtree_root.num_children--;
```

```
                    //left child changed, so recompute subtree height
                    subtree_root.update_height(nodes);
                }
            } else if (key > subtree_root.key) {
                nodes_visited = do_remove(nodes_visited,
                     subtree_root.right_index, key, value, found_key);
                if (found_key) {
                    //found the desired node and delete it
                    subtree_root.num_children--;
                    //right child changed, so recompute subtree height
                    subtree_root.update_height(nodes);
                }
            } else if (key == subtree_root.key) {
                //found key, remove the node
                found_key = true;
                value = subtree_root.value;
                remove_node(subtree_root_index);
            } else {
                throw std::logic_error("Unexpected compare result");
            }
        }
        return nodes_visited;
    }
    void write_subtree_buffer(size_t subtree_root_index,
                          CDAL<std::string>& buffer_lines,
                          size_t root_line_index,
                          size_t lbound_line_index /*inclusive*/,
                          size_t ubound_line_index /*exclusive*/) const
    {
        Node subtree_root = nodes[subtree_root_index];
        std::ostringstream oss;
        //print the node
        //todo: fix this to only print the key
        oss << "[" << subtree_root.key.raw() << "]";
        buffer_lines[root_line_index] += oss.str();
        //print the right descendents
        if (subtree_root.right_index > 0) {
            //at least 1 right child
            size_t top_dashes = 1;
            Node const& right_child = nodes[subtree_root.right_index];
            if (right_child.left_index > 0) {
                //right child has at least 1 left child
                Node const& right_left_child = nodes[right_child.left_index];
                top_dashes += 2 * (1 + right_left_child.num_children);
            }
            size_t top_line_index = root_line_index - 1;
            while (top_line_index >= root_line_index - top_dashes)
                buffer_lines[top_line_index--] += "| ";
            size_t right_child_line_index = top_line_index;
            buffer_lines[top_line_index--] += "+--";
            while (top_line_index >= lbound_line_index)
                buffer_lines[top_line_index--] += "  ";
```

```
233             write_subtree_buffer(subtree_root.right_index,
234                                  buffer_lines,
235                                  right_child_line_index,
236                                  lbound_line_index,
237                                  root_line_index);
238         }
239         //print the left descendents
240         if (subtree_root.left_index > 0) {
241             //at least 1 left child
242             size_t bottom_dashes = 1;
243             Node const& left_child = nodes[subtree_root.left_index];
244             if (left_child.right_index > 0) {
245                 //left child has at least 1 right child
246                 Node const& left_right_child = nodes[left_child.right_index];
247                 bottom_dashes += 2 * (1 + left_right_child.num_children);
248             }
249             size_t bottom_line_index = root_line_index + 1;
250             while (bottom_line_index <= root_line_index + bottom_dashes)
251                 buffer_lines[bottom_line_index++] += "| ";
252             size_t left_child_line_index = bottom_line_index;
253             buffer_lines[bottom_line_index++] += "+--";
254             while (bottom_line_index < ubound_line_index)
255                 buffer_lines[bottom_line_index++] += " ";
256             write_subtree_buffer(subtree_root.left_index,
257                                  buffer_lines,
258                                  left_child_line_index,
259                                  root_line_index + 1,
260                                  ubound_line_index);
261         }
262     }
263     void add_node_to_free_tree(size_t node_index) {
264         nodes[node_index].disable_and_adopt_free_tree(free_index);
265         free_index = node_index;
266     }
267     size_t procure_node(Key const& key, Value const& value) {
268         //updates the free index to the first free node's left child (while
                transforming that first free
269         //node to an enabled node with the specified key/value) and returns the
                index of what was the last
270         //free index
271         size_t node_index = free_index;
272         free_index = nodes[free_index].left_index;
273         Node& n = nodes[node_index];
274         n.reset_and_enable(key, value);
275         return node_index;
276     }
277     virtual int insert_at_leaf(size_t nodes_visited, //starts at 0 when this
            function is first called (ie does not include current node visitation)
278                                size_t& subtree_root_index,
279                                Key const& key,
280                                Value const& value,
281                                bool& found_key)
```

```
282         {
283             if (subtree_root_index == 0) {
284                 //key not found
285                 subtree_root_index = procure_node(key, value);
286             } else {
287                 //parent was not a leaf
288                 //keep going down to the base of the tree
289                 Node& subtree_root = nodes[subtree_root_index];
290                 ++nodes_visited;
291                 if (key < subtree_root.key) {
292                     nodes_visited = insert_at_leaf(nodes_visited,
                            subtree_root.left_index, key, value, found_key);
293                     if ( ! found_key) {
294                         //given key is unique to the tree, so a new node was added
295                         subtree_root.num_children++;
296                         subtree_root.update_height(nodes);
297                     }
298                 } else if (key > subtree_root.key) {
299                     nodes_visited = insert_at_leaf(nodes_visited,
                            subtree_root.right_index, key, value, found_key);
300                     if ( ! found_key) {
301                         //given key is unique to the tree, so a new node was added
302                         subtree_root.num_children++;
303                         subtree_root.update_height(nodes);
304                     }
305                 } else if (key == subtree_root.key) {
306                     //found key, replace the value
307                     subtree_root.value = value;
308                     found_key = true;
309                 } else {
310                     throw std::logic_error("Unexpected compare result");
311                 }
312             }
313             return nodes_visited;
314         }
315         void rotate_left(size_t& subtree_root_index) {
316             Node& subtree_root = nodes[subtree_root_index];
317             size_t right_child_index = subtree_root.right_index;
318             Node& right_child = nodes[right_child_index];
319
320             //original root adopts the right child's left subtree
321             subtree_root.right_index = right_child.left_index;
322             //original root adopted a subtree (whose height did not change), so
                    update its height
323             subtree_root.update_height(nodes);
324
325             //right child adopts original root and its children
326             right_child.left_index = subtree_root_index;
327             //right child (new root) adopted the original root (whose height has
                    been updated), so update its height
328             right_child.update_height(nodes);
```

```
329             //since right child took the subtree root's place, it has the same
                    number of children as the original root
330             right_child.num_children = subtree_root.num_children;
331
332             //root has new children, so update that counter (done after changing the
                    right child's children counter
333             //because that depends on the original root's counter)
334             subtree_root.num_children = 0;
335             if (subtree_root.left_index != 0)
336                 subtree_root.num_children += 1 +
                        nodes[subtree_root.left_index].num_children;
337             if (subtree_root.right_index != 0)
338                 subtree_root.num_children += 1 +
                        nodes[subtree_root.right_index].num_children;
339
340             //set the right child as the new root
341             subtree_root_index = right_child_index;
342         }
343         void rotate_right(size_t& subtree_root_index) {
344             Node& subtree_root = nodes[subtree_root_index];
345             size_t left_child_index = subtree_root.left_index;
346             Node& left_child = nodes[left_child_index];
347
348             //original root adopts the left child's right subtree
349             subtree_root.left_index = left_child.right_index;
350             //original root adopted a subtree (whose height did not change), so
                    update its height
351             subtree_root.update_height(nodes);
352
353             //left child adopts original root and its children
354             left_child.right_index = subtree_root_index;
355             //left child (new root) adopted the original root (whose height has been
                    updated), so update its height
356             left_child.update_height(nodes);
357             //since left child took the subtree root's place, it has the same number
                    of children as the original root
358             left_child.num_children = subtree_root.num_children;
359
360             //root has new children, so update that counter (done after changing the
                    left child's children counter
361             //because that depends on the original root's counter)
362             subtree_root.num_children = 0;
363             if (subtree_root.left_index != 0)
364                 subtree_root.num_children += 1 +
                        nodes[subtree_root.left_index].num_children;
365             if (subtree_root.right_index != 0)
366                 subtree_root.num_children += 1 +
                        nodes[subtree_root.right_index].num_children;
367
368             //set the left child as the new root
369             subtree_root_index = left_child_index;
370         }
```

```
371         int do_search(size_t nodes_visited, //starts at 0 when this function is
                first called (ie does not include current node visitation)
372                 size_t subtree_root_index,
373                 Key const& key,
374                 Value& value,
375                 bool& found_key) const
376         {
377             if (subtree_root_index != 0) {
378                 Node const& subtree_root = nodes[subtree_root_index];
379                 ++nodes_visited;
380                 if (key < subtree_root.key) {
381                     nodes_visited = do_search(nodes_visited, subtree_root.left_index,
                            key, value, found_key);
382                 } else if (key > subtree_root.key) {
383                     nodes_visited = do_search(nodes_visited,
                            subtree_root.right_index, key, value, found_key);
384                 } else if (key == subtree_root.key) {
385                     //found key, replace the value
386                     value = subtree_root.value;
387                     found_key = true;
388                 } else {
389                     throw std::logic_error("Unexpected compare result");
390                 }
391             }
392             return nodes_visited;
393         }
394         void prepare_cluster_distribution(size_t subtree_root_index,
395                                           size_t curr_height, //includes the height of
                                                the current node, ie assumes current node
                                                exists
396                                           size_t cluster_counter[])
397         {
398             Node const& subtree_root = nodes[subtree_root_index];
399             if ( ! subtree_root.left_index && ! subtree_root.right_index)
400                 //at a leaf node
401                 cluster_counter[curr_height]++;
402             else {
403                 if (subtree_root.left_index)
404                     prepare_cluster_distribution(subtree_root.left_index, curr_height
                            + 1, cluster_counter);
405                 if (subtree_root.right_index)
406                     prepare_cluster_distribution(subtree_root.right_index,
                            curr_height + 1, cluster_counter);
407             }
408         }
409
410         void remove_ith_node_inorder(size_t& subtree_root_index,
411                                      size_t& ith_node_to_delete,
412                                      Key& key)
413         {
414             Node& subtree_root = nodes[subtree_root_index];
415             if (subtree_root.left_index)
```

```
416                 remove_ith_node_inorder(subtree_root.left_index, ith_node_to_delete,
                        key);
417             if (ith_node_to_delete == 0)
418                 //deleted node in child subtree; nothing more to do
419                 return;
420             if (--ith_node_to_delete == 0) {
421                 //delete the current node
422                 value_type dummy_val;
423                 remove(subtree_root.key.raw_copy(), dummy_val);
424                 key = subtree_root.key;
425                 return;
426             }
427             if (subtree_root.right_index)
428                 remove_ith_node_inorder(subtree_root.right_index,
                        ith_node_to_delete, key);
429         }
430
431     public:
432         /*
433             The constructor will allocate an array of capacity (binary
434             tree) nodes. Then make a chain from all the nodes (e.g.,
435             make node 2 the left child of node 1, make node 3 the left
436             child of node 2, &c. this is the initial free list.
437         */
438         BST(size_t capacity):
439             curr_capacity(capacity)
440         {
441             if (capacity == 0) {
442                 throw std::domain_error("capacity must be at least 1");
443             }
444             nodes = new Node[capacity + 1];
445             clear();
446         }
447         /*
448             if there is space available, adds the specified key/value-pair to the
                    tree
449             and returns the number of nodes visited, V; otherwise returns -1 * V. If
                    an
450             item already exists in the tree with the same key, replace its value.
451         */
452         virtual int insert(key_type const& key, value_type const& value) {
453             if (size() == capacity())
454                 //no more space
455                 return -1 * size();
456             bool found_key = false;
457             Key k(key);
458             Value v(value);
459             int nodes_visited = insert_at_leaf(0, root_index, k, v, found_key);
460             if (_DEBUG_)
461                 this->nodes[this->root_index].validate_children_count_recursive(this->nodes);
462             return nodes_visited;
463         }
```

```
464         /*
465             if there is an item matching key, removes the key/value-pair from the
                tree, stores
466             it's value in value, and returns the number of probes required, V;
                otherwise returns -1 * V.
467         */
468         virtual int remove(key_type const& key, value_type& value) {
469             if (is_empty())
470                 return 0;
471             bool found_key = false;
472             Key k(key);
473             Value v(value);
474             int nodes_visited = do_remove(0, root_index, k, v, found_key);
475             if (_DEBUG_)
476                 this->nodes[this->root_index].validate_children_count_recursive(this->nodes);
477             if (found_key)
478                 value = v.raw_copy();
479             return found_key ? nodes_visited : -1 * nodes_visited;
480         }
481         /*
482             if there is an item matching key, stores it's value in value, and
                returns the number
483             of nodes visited, V; otherwise returns -1 * V. Regardless, the item
                remains in the tree.
484         */
485         virtual int search(key_type const& key, value_type& value) {
486             if (is_empty())
487                 return 0;
488             bool found_key = false;
489             Key k(key);
490             Value v(value);
491             int nodes_visited = do_search(0, root_index, k, v, found_key);
492             if (found_key)
493                 value = v.raw_copy();
494             return found_key ? nodes_visited : -1 * nodes_visited;
495         }
496         /*
497             removes all items from the map
498         */
499         virtual void clear() {
500             //Since I use size_t to hold the node indices, I make the node array
501             //1-based, with child index of 0 indicating that the current node is a
                    leaf
502             for (size_t i = 1; i != capacity(); ++i)
503                 nodes[i].disable_and_adopt_free_tree(i + 1);
504             free_index = 1;
505             root_index = 0;
506         }
507         /*
508             returns true IFF the map contains no elements.
509         */
510         virtual bool is_empty() const {
```

```
511             return size() == 0;
512         }
513         /*
514             returns the number of slots in the backing array.
515         */
516         virtual size_t capacity() const {
517             return curr_capacity;
518         }
519         /*
520             returns the number of items actually stored in the tree.
521         */
522         virtual size_t size() const {
523             if (root_index == 0) return 0;
524             Node const& root = nodes[root_index];
525             return 1 + root.num_children;
526         }
527         /*
528             [not a regular BST operation, but specific to this implementation]
529             returns the tree's load factor: load = size / capacity.
530         */
531         virtual double load() const {
532             return static_cast<double>(size()) / capacity();
533         }
534         /*
535             prints the tree in the following format:
536             +--[tiger]
537             |  |
538             |  |  +--[panther]
539             |  |  |
540             |  +--[ocelot]
541             |     |
542             |     +--[lion]
543             |
544             [leopard]
545             |
546             |     +--[house cat]
547             |     |
548             |  +--[cougar]
549             |  |
550             +--[cheetah]
551                |
552                +--[bobcat]
553         */
554         virtual std::ostream& print(std::ostream& out) const {
555             if (is_empty())
556                 return out;
557             size_t num_lines = size() * 2 - 1;
558             //use CDAL here so we can print really super-huge trees where the write
                    buffer doesn't fit in memory
559             CDAL<std::string> buffer_lines(100000);
560             for(size_t i = 0; i <= num_lines; ++i)
561                 buffer_lines.push_back("");
```

```
562             Node const& root = nodes[root_index];
563             size_t root_line_index = 1;
564             if (root.right_index) {
565                 root_line_index += 2 * (1 + nodes[root.right_index].num_children);
566             }
567             write_subtree_buffer(root_index, buffer_lines, root_line_index, 1,
                    num_lines + 1);
568             for (size_t i = 1; i <= num_lines; ++i)
569                 out << buffer_lines[i] << std::endl;
570             return out;
571         }
572
573         /*
574             returns a list indicating the number of leaf nodes at each height (since
                    the RBST doesn't exhibit
575             true clustering, but can have degenerate branches).
576         */
577         virtual priority_queue<hash_utils::ClusterInventory> cluster_distribution()
                {
578             //use an array to count cluster instances, then feed those to a priority
                    queue and return it.
579             priority_queue<ClusterInventory> cluster_pq;
580             if (is_empty()) return cluster_pq;
581             size_t max_height = nodes[root_index].height;
582             size_t cluster_counter[max_height + 1];
583             for (size_t i = 0; i <= max_height; ++i)
584                 cluster_counter[i] = 0;
585             prepare_cluster_distribution(root_index, 1, cluster_counter);
586             for (size_t i = 1; i <= max_height; ++i)
587                 if (cluster_counter[i] > 0) {
588                     ClusterInventory cluster{i, cluster_counter[i]};
589                     cluster_pq.add_to_queue(cluster);
590                 }
591             return cluster_pq;
592         }
593
594         /*
595             generate a random number, R, (1,size), and starting with the root (node
                    1), do an in-order
596             traversal to find the R-th occupied node; remove that node (adjusting
                    its children accordingly),
597             and return its key.
598
599             ***XXX: this likely contains a bug when using const char* keys in that
                    we'll be returning a dangling pointer!!!***
600         */
601         virtual key_type remove_random() {
602             if (size() == 0) throw std::logic_error("Cant remove from an empty map");
603             size_t ith_node_to_delete = 1 + rand_i(size());
604             Key key;
605             remove_ith_node_inorder(root_index, ith_node_to_delete, key);
606             key_type ret = key.raw_copy();
```

```
607             return ret;
608         }
609     };
610 }
611
612 #endif
```

part4_bonus/source/avl.h

part4_bonus/source/avl.h

```
#ifndef _AVL_H_
#define _AVL_H_

#include <cstdlib>
#include <sstream>
#include "../../common/CDAL.h"
#include "../../common/common.h"
#include "../../common/priority_queue.h"
#include "../../part4/source/bst.h"

namespace cop3530 {
    template<typename key_type,
             typename value_type>
    class AVL: public BST<key_type, value_type> {
    /*
        The trick to AVL is to perform standard BST operations, but wrap recursive methods that might unbalance
        the tree with methods that rebalance the tree after performing those operations. Thus the balance factor
        of any given node stays within [-1, 1]. To that end we simply inherit from a BST base class that tracks
        changes in subtree height and overwrite the needed virtual methods.
    */
    private:
        using super = BST<key_type, value_type>;
        using typename super::Node;
        typedef hash_utils::Key<key_type> Key;
        typedef hash_utils::Value<value_type> Value;
        int insert_at_leaf(size_t nodes_visited,
                           size_t& subtree_root_index,
                           Key const& key,
                           Value const& value,
                           bool& found_key)
        {
            nodes_visited = super::insert_at_leaf(nodes_visited, subtree_root_index, key, value, found_key);
            balance(subtree_root_index);
            return nodes_visited;
        }
        size_t remove_smallest_key_node_index(size_t& subtree_root_index) {
            size_t smallest_key_node_index =
                super::remove_smallest_key_node_index(subtree_root_index);
            balance(subtree_root_index);
            return smallest_key_node_index;
        }
        size_t remove_largest_key_node_index(size_t& subtree_root_index) {
```

```
42              size_t largest_key_node_index =
                    super::remove_largest_key_node_index(subtree_root_index);
43              balance(subtree_root_index);
44              return largest_key_node_index;
45          }
46          int do_remove(size_t nodes_visited, //starts at 0 when this function is
                first called (ie does not include current node visitation)
47                        size_t& subtree_root_index,
48                        Key const& key,
49                        Value& value,
50                        bool& found_key)
51          {
52              nodes_visited = super::do_remove(nodes_visited, subtree_root_index, key,
                    value, found_key);
53              balance(subtree_root_index);
54              return nodes_visited;
55          }
56          void balance(size_t& subtree_root_index) {
57              if (subtree_root_index == 0) return;
58              Node& root = this->nodes[subtree_root_index];
59              int root_bal_fact = root.balance_factor(this->nodes);
60              if (root_bal_fact == -2) {
61                  //right subtree is too heavy
62                  size_t& right_index = root.right_index;
63                  Node& right_child = this->nodes[right_index];
64                  switch(right_child.balance_factor(this->nodes)) {
65                  case 1:
66                      //right left
67                      this->rotate_right(right_index);
68                      this->rotate_left(subtree_root_index);
69                      break;
70                  case -1:
71                  case 0:
72                      //right right
73                      this->rotate_left(subtree_root_index);
74                      break;
75                  default:
76                      throw std::domain_error(std::string("Unexpected balance factor
                            with heavy right subtree: ")
77                                              +
                                                    std::to_string(right_child.balance_factor(this->nodes)));
78                  }
79              } else if (root_bal_fact == 2) {
80                  //left subtree is too heavy
81                  size_t& left_index = root.left_index;
82                  Node& left_child = this->nodes[left_index];
83                  switch(left_child.balance_factor(this->nodes)) {
84                  case -1:
85                      //left right
86                      this->rotate_left(left_index);
87                      this->rotate_right(subtree_root_index);
88                      break;
```

```
89                case 1:
90                case 0:
91                    //left left
92                    this->rotate_right(subtree_root_index);
93                    break;
94                default:
95                    throw std::domain_error(std::string("Unexpected balance factor
                          with heavy left subtree: ")
96                                            +
                                                std::to_string(left_child.balance_factor(this->nodes)));
97                }
98            } else if (std::abs(root_bal_fact > 2)) {
99                throw std::domain_error(std::string("Unexpected balance factor when
                       checking for heavy subtree: ")
100                                       + std::to_string(root_bal_fact));
101           }
102       }
103       void do_validate_avl_balance(size_t subtree_root_index) const {
104           if (_DEBUG_) {
105               if (subtree_root_index == 0) return;
106               Node const& n = this->nodes[subtree_root_index];
107               if (abs(n.balance_factor(this->nodes)) > 1)
108                   throw std::domain_error("Unexpected unbalanced tree while
                          checking balance factor of all tree nodes");
109               do_validate_avl_balance(n.left_index);
110               do_validate_avl_balance(n.right_index);
111           }
112       }
113       void validate_avl_balance() {
114           if (_DEBUG_)
115               do_validate_avl_balance(this->root_index);
116       }
117   public:
118       AVL(size_t capacity): super(capacity) {}
119       /*
120           if there is space available, adds the specified key/value-pair to the
                tree
121           and returns the number of nodes visited, V; otherwise returns -1 * V. If
                an
122           item already exists in the tree with the same key, replace its value.
123       */
124       int insert(key_type const& key, value_type const& value) {
125           if (this->size() == this->capacity())
126               //no more space
127               return -1 * this->size();
128           bool found_key = false;
129           Key k(key);
130           Value v(value);
131           int nodes_visited = insert_at_leaf(0, this->root_index, k, v, found_key);
132           validate_avl_balance();
133           if (_DEBUG_)
134               this->nodes[this->root_index].validate_children_count_recursive(this->nodes);
```

```
135             return nodes_visited;
136         }
137         /*
138             if there is an item matching key, removes the key/value-pair from the
                    tree, stores
139             it's value in value, and returns the number of probes required, V;
                    otherwise returns -1 * V.
140         */
141         int remove(key_type const& key, value_type& value) {
142             if (this->is_empty())
143                 return 0;
144             bool found_key = false;
145             Key k(key);
146             Value v(value);
147             int nodes_visited = do_remove(0, this->root_index, k, v, found_key);
148             validate_avl_balance();
149             if (_DEBUG_)
150                 this->nodes[this->root_index].validate_children_count_recursive(this->nodes);
151             if (found_key)
152                 value = v.raw_copy();
153             return found_key ? nodes_visited : -1 * nodes_visited;
154         }
155     };
156 }
157
158 #endif
```