SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

SSLL(const SSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

SSLL& operator=(const SSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
  - In this case we pass the element to push_back(), which can do O(1) insert
  - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail

- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list

- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

Iterator Methods

explicit SSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Iter(const SSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

Const Iterator Methods

explicit SSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list

- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Const_Iter(const SSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

Part I: Hashmap with Open Addressing

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

SSLL(const SSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

SSLL& operator=(const SSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
  - In this case we pass the element to push_back(), which can do O(1) insert
  - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail

- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list

- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

Iterator Methods

explicit SSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Iter(const SSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

Const Iterator Methods

explicit SSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list

- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Const_Iter(const SSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

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_

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

namespace cop3530 {
    double lg(size_t i) {
        return std::log(i) / std::log(2);
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        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;
                }
            };
        };
        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;
        }
        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) {
                    //make capacity a power of 2, greater than the minimum capacity
                    return 1 << static_cast<size_t>(std::ceil(lg(min_capacity)));
                }
            };
```

```
47              struct compare_functor {
48                  int operator()(const char* a, const char* b) const {
49                      int cmp = strcmp(a, b);
50                      return (cmp < 0 ? -1 :
51                                      (cmp > 0 ? 1 : 0));
52                  }
53                  int operator()(double a, double b) const {
54                      return (a < b ? -1 :
55                                      (a > b ? 1 : 0));
56                  }
57                  int operator()(std::string const& a, std::string const& b) const {
58                      return (a < b ? -1 :
59                                      (a > b ? 1 : 0));
60                  }
61                  int operator()(int a, int b) const {
62                      return (a < b ? -1 :
63                                      (a > b ? 1 : 0));
64                  }
65              };
66              namespace primary_hashes {
67                  struct hash_basic {
68                  //this is such a stupid hash method, but unlike my pathetic attempts
                        at implementing
69                  //various other hashing methods, it works and is generalizable to
                        all the required key
70                  //types. together with double hashing it should make for a passable
                        hashing routine.
71                  public:
72                      size_t operator()(const char* key) const {
73                          return str_to_numeric(key);
74                      }
75                      size_t operator()(double key) const {
76                          return static_cast<size_t>(std::fmod(key, max_size_t));
77                      }
78                      size_t operator()(int key) const {
79                          return static_cast<size_t>(key);
80                      }
81                      size_t operator()(std::string const& key) const {
82                          const char* c_key = key.c_str();
83                          return operator()(c_key);
84                      }
85                  };
86              }
87              namespace secondary_hashes {
88                  struct linear_probe {
89                      bool changes_with_probe_attempt() const {
90                          return false;
91                      }
92                      size_t operator()(const char* key, size_t probe_attempt) const {
93                          return 1;
94                      }
95                  };
```

```
96                  struct quadratic_probe {
97                      bool changes_with_probe_attempt() const {
98                          return true;
99                      }
100                     size_t operator()(const char* key, size_t probe_attempt) const {
101                         return probe_attempt;
102                     }
103                 };
104                 struct hash_double {
105                 private:
106                     size_t hash_numeric(size_t numeric) const {
107                         size_t hash = numeric % 97; //simple modulus using a prime
                                number (from algorithms in c++)
108                         //the second hash may not be zero (will cause an infinite
                                loop).
109                         //also, hash must be relatively prime to map_capacity so that
                                every slot can be hit.
110                         //since map capacity is a power of two if we use the capacity
                                planner functor,
111                         //both properties are attainable by adding one to the hash if
                                it is even (despite what my
112                         //7th grade algebra teacher attempted to teach me, I
                                stubbournly consider zero to be an even
113                         //integer despite no formal training in number theory)
114                         bool is_even = (hash & 1) == 0;
115                         if (is_even)
116                             ++hash;
117                         return hash;
118                     }
119                 public:
120                     bool changes_with_probe_attempt() const {
121                         return false;
122                     }
123                     size_t operator()(const char* key, size_t unused) const {
124                         size_t numeric = str_to_numeric(key);
125                         return hash_numeric(numeric);
126                     }
127                     size_t operator()(double key, size_t unused) const {
128                         return hash_numeric(key);
129                     }
130                     size_t operator()(int key, size_t unused) const {
131                         return hash_numeric(key);
132                     }
133                     size_t operator()(std::string key, size_t unused) const {
134                         const char* c_key = key.c_str();
135                         return operator()(c_key, unused);
136                     }
137                 };
138             }
139         }
140     }
141 }
```

```
142
143 std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory
        const& rhs) {
144     out << "Cluster{size=" << rhs.cluster_size << ", instances=" <<
            rhs.num_instances << "}";
145     return out;
146 }
147
148 #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 takes a simple singly linked list containing clusters and exposes
    //a method (get_next_item) which returns the clusters is order of ascending size
    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;
            }
        }
```

```
45      public:
46          //take a linked list of cluster descriptors and add each to the priority
                queue
47          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
48              T empty_item;
49              tree.push_back(empty_item);
50          }
51          priority_queue(priority_queue const& src) {
52              tree = src.tree;
53              num_items = src.num_items;
54          }
55          T get_next_item() {
56              std::swap(tree[1], tree[num_items]);
57              T ret = tree[num_items--];
58              fix_down();
59              return ret;
60          }
61          void add_to_queue(T const& item) {
62              tree.push_back(item);
63              num_items++;
64              fix_up(num_items);
65          }
66          size_t size() {
67              return num_items;
68          }
69          bool empty() {
70              return num_items == 0;
71          }
72      };
73  }
74
75  #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"

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 probes_required;
            for (probes_required = 0; probes_required != M; ++probes_required) {
                size_t slot_index = (hash_val + probe(probes_required)) % M;
                if (slots[slot_index].is_occupied) {
                    if (slots[slot_index].key == key) {
                        //found the key
                        break;
                    }
                } else
```

```
47                  //found unoccupied slot
48                  break;
49              }
50              return probes_required;
51          }
52          //all backing array manipulations should go through the following two
                methods
53          void insert_at_index(key_type const& key, value_type const& value, size_t
                index) {
54              Slot& s = slots[index];
55              s.key = key;
56              s.value = value;
57              if ( ! s.is_occupied) {
58                  s.is_occupied = true;
59                  ++num_occupied_slots;
60              }
61          }
62          value_type remove_at_index(size_t index) {
63              Slot& s = slots[index];
64              if (s.is_occupied) {
65                  s.is_occupied = false;
66                  --num_occupied_slots;
67              }
68              return s.value;
69          }
70      public:
71          HashMapOpenAddressing(size_t const min_capacity)
72          {
73              if (min_capacity == 0) {
74                  throw std::domain_error("min_capacity must be at least 1");
75              }
76              cop3530::hash_utils::functors::map_capacity_planner capacity_planner;
77              curr_capacity = capacity_planner(min_capacity); //make capacity a power
                    of 2, greater than the minimum capacity
78              slots = new Slot[curr_capacity];
79          }
80          ~HashMapOpenAddressing() {
81              delete slots;
82          }
83          /*
84              if there is space available, adds the specified key/value-pair to the
                    hash map and returns true; otherwise
85              returns false. If an item already exists in the map with the same key,
                    replace its value.
86          */
87          bool insert(key_type const& key, value_type const& value) {
88              size_t M = capacity();
89              if (M == size())
90                  return false;
91              size_t probes_required = search_internal(key);
92              size_t index = (hash(key) + probe(probes_required)) % M;
93              insert_at_index(key, value, index);
```

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

```
145         /*
146             returns true IFF the map contains no elements.
147         */
148         bool is_empty() {
149             return size() == 0;
150         }
151         /*
152             returns the number of slots in the map.
153         */
154         size_t capacity() {
155             return curr_capacity;
156         }
157         /*
158             returns the number of items actually stored in the map.
159         */
160         size_t size() {
161             return num_occupied_slots;
162         }
163         /*
164             returns the map's load factor (size = load * capacity).
165         */
166         double load() {
167             return static_cast<double>(size()) / capacity();
168         }
169         /*
170             inserts into the ostream, the backing array's contents in sequential
                order.
171             Empty slots shall be denoted by a hyphen, non-empty slots by that item's
172             key. [This function will be used for debugging/monitoring].
173         */
174         std::ostream& print(std::ostream& out) {
175             size_t cap = capacity();
176             out << '[';
177             for (size_t i = 0; i != cap; ++i) {
178                 if (slots[i].is_occupied) {
179                     out << slots[i].key;
180                 } else {
181                     out << "-";
182                 }
183                 if (i + 1 < cap)
184                     out << '|';
185             }
186             out << ']';
187             return out;
188         }
189 
190     };
191 }
192 
193 #endif
```

Part II: Hashmap with Buckets

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

SSLL(const SSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

SSLL& operator=(const SSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
  - In this case we pass the element to push_back(), which can do O(1) insert
  - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail

- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list

- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

Iterator Methods

explicit SSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Iter(const SSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

Const Iterator Methods

explicit SSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list

- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Const_Iter(const SSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

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_

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

namespace cop3530 {
    double lg(size_t i) {
        return std::log(i) / std::log(2);
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        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;
                }
            };
        };
        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;
        }
        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) {
                    //make capacity a power of 2, greater than the minimum capacity
                    return 1 << static_cast<size_t>(std::ceil(lg(min_capacity)));
                }
            };
```

```
47          struct compare_functor {
48              int operator()(const char* a, const char* b) const {
49                  int cmp = strcmp(a, b);
50                  return (cmp < 0 ? -1 :
51                                  (cmp > 0 ? 1 : 0));
52              }
53              int operator()(double a, double b) const {
54                  return (a < b ? -1 :
55                                  (a > b ? 1 : 0));
56              }
57              int operator()(std::string const& a, std::string const& b) const {
58                  return (a < b ? -1 :
59                                  (a > b ? 1 : 0));
60              }
61              int operator()(int a, int b) const {
62                  return (a < b ? -1 :
63                                  (a > b ? 1 : 0));
64              }
65          };
66          namespace primary_hashes {
67              struct hash_basic {
68              //this is such a stupid hash method, but unlike my pathetic attempts
                    at implementing
69              //various other hashing methods, it works and is generalizable to
                    all the required key
70              //types. together with double hashing it should make for a passable
                    hashing routine.
71              public:
72                  size_t operator()(const char* key) const {
73                      return str_to_numeric(key);
74                  }
75                  size_t operator()(double key) const {
76                      return static_cast<size_t>(std::fmod(key, max_size_t));
77                  }
78                  size_t operator()(int key) const {
79                      return static_cast<size_t>(key);
80                  }
81                  size_t operator()(std::string const& key) const {
82                      const char* c_key = key.c_str();
83                      return operator()(c_key);
84                  }
85              };
86          }
87          namespace secondary_hashes {
88              struct linear_probe {
89                  bool changes_with_probe_attempt() const {
90                      return false;
91                  }
92                  size_t operator()(const char* key, size_t probe_attempt) const {
93                      return 1;
94                  }
95              };
```

```
96                  struct quadratic_probe {
97                      bool changes_with_probe_attempt() const {
98                          return true;
99                      }
100                     size_t operator()(const char* key, size_t probe_attempt) const {
101                         return probe_attempt;
102                     }
103                 };
104                 struct hash_double {
105                 private:
106                     size_t hash_numeric(size_t numeric) const {
107                         size_t hash = numeric % 97; //simple modulus using a prime
                                number (from algorithms in c++)
108                         //the second hash may not be zero (will cause an infinite
                                loop).
109                         //also, hash must be relatively prime to map_capacity so that
                                every slot can be hit.
110                         //since map capacity is a power of two if we use the capacity
                                planner functor,
111                         //both properties are attainable by adding one to the hash if
                                it is even (despite what my
112                         //7th grade algebra teacher attempted to teach me, I
                                stubbournly consider zero to be an even
113                         //integer despite no formal training in number theory)
114                         bool is_even = (hash & 1) == 0;
115                         if (is_even)
116                             ++hash;
117                         return hash;
118                     }
119                 public:
120                     bool changes_with_probe_attempt() const {
121                         return false;
122                     }
123                     size_t operator()(const char* key, size_t unused) const {
124                         size_t numeric = str_to_numeric(key);
125                         return hash_numeric(numeric);
126                     }
127                     size_t operator()(double key, size_t unused) const {
128                         return hash_numeric(key);
129                     }
130                     size_t operator()(int key, size_t unused) const {
131                         return hash_numeric(key);
132                     }
133                     size_t operator()(std::string key, size_t unused) const {
134                         const char* c_key = key.c_str();
135                         return operator()(c_key, unused);
136                     }
137                 };
138             }
139         }
140     }
141 }
```

```
142
143 std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory
        const& rhs) {
144     out << "Cluster{size=" << rhs.cluster_size << ", instances=" <<
            rhs.num_instances << "}";
145     return out;
146 }
147
148 #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 takes a simple singly linked list containing clusters and exposes
    //a method (get_next_item) which returns the clusters is order of ascending size
    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;
            }
        }
```

```
45      public:
46          //take a linked list of cluster descriptors and add each to the priority
                queue
47          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
48              T empty_item;
49              tree.push_back(empty_item);
50          }
51          priority_queue(priority_queue const& src) {
52              tree = src.tree;
53              num_items = src.num_items;
54          }
55          T get_next_item() {
56              std::swap(tree[1], tree[num_items]);
57              T ret = tree[num_items--];
58              fix_down();
59              return ret;
60          }
61          void add_to_queue(T const& item) {
62              tree.push_back(item);
63              num_items++;
64              fix_up(num_items);
65          }
66          size_t size() {
67              return num_items;
68          }
69          bool empty() {
70              return num_items == 0;
71          }
72      };
73  }
74
75  #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 "../../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
```

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

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

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

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

Part III: Parameterizable Hashmap with Open Addressing

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

SSLL(const SSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

SSLL& operator=(const SSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
  - In this case we pass the element to push_back(), which can do O(1) insert
  - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail

- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list

- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

Iterator Methods

explicit SSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Iter(const SSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

Const Iterator Methods

explicit SSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list

- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Const_Iter(const SSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

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: 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 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_

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

namespace cop3530 {
    double lg(size_t i) {
        return std::log(i) / std::log(2);
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        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;
                }
            };
        };
        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;
        }
        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) {
                    //make capacity a power of 2, greater than the minimum capacity
                    return 1 << static_cast<size_t>(std::ceil(lg(min_capacity)));
                }
            };
```

```
47          struct compare_functor {
48              int operator()(const char* a, const char* b) const {
49                  int cmp = strcmp(a, b);
50                  return (cmp < 0 ? -1 :
51                                  (cmp > 0 ? 1 : 0));
52              }
53              int operator()(double a, double b) const {
54                  return (a < b ? -1 :
55                                  (a > b ? 1 : 0));
56              }
57              int operator()(std::string const& a, std::string const& b) const {
58                  return (a < b ? -1 :
59                                  (a > b ? 1 : 0));
60              }
61              int operator()(int a, int b) const {
62                  return (a < b ? -1 :
63                                  (a > b ? 1 : 0));
64              }
65          };
66          namespace primary_hashes {
67              struct hash_basic {
68              //this is such a stupid hash method, but unlike my pathetic attempts
                    at implementing
69              //various other hashing methods, it works and is generalizable to
                    all the required key
70              //types. together with double hashing it should make for a passable
                    hashing routine.
71              public:
72                  size_t operator()(const char* key) const {
73                      return str_to_numeric(key);
74                  }
75                  size_t operator()(double key) const {
76                      return static_cast<size_t>(std::fmod(key, max_size_t));
77                  }
78                  size_t operator()(int key) const {
79                      return static_cast<size_t>(key);
80                  }
81                  size_t operator()(std::string const& key) const {
82                      const char* c_key = key.c_str();
83                      return operator()(c_key);
84                  }
85              };
86          }
87          namespace secondary_hashes {
88              struct linear_probe {
89                  bool changes_with_probe_attempt() const {
90                      return false;
91                  }
92                  size_t operator()(const char* key, size_t probe_attempt) const {
93                      return 1;
94                  }
95              };
```

```
96                  struct quadratic_probe {
97                      bool changes_with_probe_attempt() const {
98                          return true;
99                      }
100                     size_t operator()(const char* key, size_t probe_attempt) const {
101                         return probe_attempt;
102                     }
103                 };
104                 struct hash_double {
105                 private:
106                     size_t hash_numeric(size_t numeric) const {
107                         size_t hash = numeric % 97; //simple modulus using a prime
                                number (from algorithms in c++)
108                         //the second hash may not be zero (will cause an infinite
                                loop).
109                         //also, hash must be relatively prime to map_capacity so that
                                every slot can be hit.
110                         //since map capacity is a power of two if we use the capacity
                                planner functor,
111                         //both properties are attainable by adding one to the hash if
                                it is even (despite what my
112                         //7th grade algebra teacher attempted to teach me, I
                                stubbournly consider zero to be an even
113                         //integer despite no formal training in number theory)
114                         bool is_even = (hash & 1) == 0;
115                         if (is_even)
116                             ++hash;
117                         return hash;
118                     }
119                 public:
120                     bool changes_with_probe_attempt() const {
121                         return false;
122                     }
123                     size_t operator()(const char* key, size_t unused) const {
124                         size_t numeric = str_to_numeric(key);
125                         return hash_numeric(numeric);
126                     }
127                     size_t operator()(double key, size_t unused) const {
128                         return hash_numeric(key);
129                     }
130                     size_t operator()(int key, size_t unused) const {
131                         return hash_numeric(key);
132                     }
133                     size_t operator()(std::string key, size_t unused) const {
134                         const char* c_key = key.c_str();
135                         return operator()(c_key, unused);
136                     }
137                 };
138             }
139         }
140     }
141 }
```

```
142
143 std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory
        const& rhs) {
144     out << "Cluster{size=" << rhs.cluster_size << ", instances=" <<
            rhs.num_instances << "}";
145     return out;
146 }
147
148 #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 takes a simple singly linked list containing clusters and exposes
    //a method (get_next_item) which returns the clusters is order of ascending size
    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;
            }
        }
```

```
45      public:
46          //take a linked list of cluster descriptors and add each to the priority
                queue
47          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
48              T empty_item;
49              tree.push_back(empty_item);
50          }
51          priority_queue(priority_queue const& src) {
52              tree = src.tree;
53              num_items = src.num_items;
54          }
55          T get_next_item() {
56              std::swap(tree[1], tree[num_items]);
57              T ret = tree[num_items--];
58              fix_down();
59              return ret;
60          }
61          void add_to_queue(T const& item) {
62              tree.push_back(item);
63              num_items++;
64              fix_up(num_items);
65          }
66          size_t size() {
67              return num_items;
68          }
69          bool empty() {
70              return num_items == 0;
71          }
72      };
73  }
74
75  #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 compare_functor = hash_utils::functors::compare_functor,
             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;
        class Key {
        private:
            key_type raw_key;
            compare_functor compare;
            primary_hash hasher1;
            secondary_hash hasher2;
            size_t hash1_val;
            size_t hash2_val;
            size_t old_map_capacity;
        public:
            bool operator==(Key const& rhs) const {
                return compare(raw_key, rhs.raw_key) == 0;
            }
            bool operator==(key_type const& rhs) const {
                return compare(raw_key, rhs) == 0;
            }
            bool operator!=(Key const& rhs) const {
                return ! operator==(rhs);
            }
            bool operator!=(key_type const& rhs) const {
                return ! operator==(rhs);
            }
            size_t hash(size_t map_capacity, size_t probe_attempt) const {
                size_t local_hash2_val;
                if (probe_attempt != 0 && hasher2.changes_with_probe_attempt())
                {
```

```
                //if the hashing function value is dependent on the probe attempt
                //(eg quadratic probing), then we need to retrieve the new value
                local_hash2_val = hasher2(raw_key, probe_attempt);
            } else {
                //otherwise we can just use the value we have stored
                local_hash2_val = hash2_val;
            }
            return (hash1_val + probe_attempt * local_hash2_val) % map_capacity;
        }
        key_type const& raw() const {
            return raw_key;
        }
        void reset(key_type const& key) {
            raw_key = key;
            size_t base_probe_attempt = 0;
            hash1_val = hasher1(key);
            hash2_val = hasher2(key, base_probe_attempt);
        }
        explicit Key(key_type key) {
            reset(key);
        }
        Key() = default;
    };
    class Value {
    private:
        value_type raw_value;
    public:
        bool operator==(Value const& rhs) const {
            return compare(raw_value, rhs.raw_value);
        }
        bool operator==(value_type const& rhs) const {
            return compare(raw_value, rhs) == 0;
        }
        value_type const& raw() const {
            return raw_value;
        }
        explicit Value(value_type value): raw_value(value) {}
        Value() = default;
    };
    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;
    /*
```

```
97              searches the map for an item matching key. returns the number of probe
                    attempts needed
98              to reach either the item or an empty slot
99          */
100         int search_internal(Key const& key) {
101             size_t M = capacity();
102             size_t probes_required;
103             for (probes_required = 0; probes_required != M; ++probes_required) {
104                 size_t slot_index = key.hash(M, probes_required);
105                 if (slots[slot_index].is_occupied) {
106                     if (slots[slot_index].item.key == key) {
107                         //found the key
108                         break;
109                     }
110                 } else
111                     //found unoccupied slot
112                     break;
113             }
114             return probes_required;
115         }
116
117         //all backing array manipulations should go through the following two
                methods
118         void insert_at_index(Key const& key, Value const& value, size_t index) {
119             Slot& s = slots[index];
120             s.item.key = key;
121             s.item.value = value;
122             if ( ! s.is_occupied) {
123                 s.is_occupied = true;
124                 ++num_occupied_slots;
125             }
126         }
127         Value const& remove_at_index(size_t index) {
128             Slot& s = slots[index];
129             if (s.is_occupied) {
130                 s.is_occupied = false;
131                 --num_occupied_slots;
132             }
133             return s.item.value;
134         }
135     public:
136         HashMapOpenAddressingGeneric(size_t const min_capacity)
137         {
138             if (min_capacity == 0) {
139                 throw std::domain_error("min_capacity must be at least 1");
140             }
141             curr_capacity = choose_capacity(min_capacity);
142             slots = new Slot[curr_capacity];
143         }
144         ~HashMapOpenAddressingGeneric() {
145             delete[] slots;
146         }
```

```
147
148         /*
149             if there is space available, adds the specified key/value-pair to the
                hash map and returns the
150             number of probes required, P; otherwise returns -1 * P. If an item
                already exists in the map
151             with the same key, replace its value.
152         */
153         int insert(key_type const& key, value_type const& value) {
154             size_t M = capacity();
155             if (M == size())
156                 return -1 * size();
157             Key k(key);
158             Value v(value);
159             size_t probes_required = search_internal(k);
160             size_t index = k.hash(M, probes_required);
161             insert_at_index(k, v, index);
162             return probes_required;
163         }
164
165         /*
166             if there is an item matching key, removes the key/value-pair from the
                map, stores it's value in
167             value, and returns the number of probes required, P; otherwise returns
                -1 * P.
168         */
169         int remove(key_type const& key, value_type& value) {
170             size_t M = capacity();
171             Key k(key);
172             size_t probes_required = search_internal(k);
173             size_t index = k.hash(M, probes_required);
174             if (slots[index].is_occupied == false || slots[index].item.key != key)
175                 //key not found
176                 return -1 * probes_required;
177             Value v = remove_at_index(index);
178             value = v.raw();
179             //remove and reinsert items until find unoccupied slot (guaranteed to
                happen since we just removed an item)
180             for (int i = 1; ; ++i) {
181                 index = k.hash(M, i);
182                 Slot const& s = slots[index];
183                 if (s.is_occupied) {
184                     remove_at_index(index);
185                     insert(s.item.key.raw(), s.item.value.raw());
186                 } else {
187                     break;
188                 }
189             }
190             return probes_required;
191         }
192
193         /*
```

```
194             if there is an item matching key, stores it's value in value, and
                    returns the
195             number of probes required, P; otherwise returns -1 * P. Regardless, the
                    item
196             remains in the map.
197         */
198         int search(key_type const& key, value_type& value) {
199             size_t M = capacity();
200             Key k(key);
201             size_t probes_required = search_internal(k);
202             size_t index = k.hash(M, probes_required);
203             if (slots[index].is_occupied == false || slots[index].item.key != key)
204                 //key not found
205                 return -1 * probes_required;
206             value = slots[index].item.value.raw();
207             return probes_required;
208         }
209
210         /*
211             removes all items from the map.
212         */
213         void clear() {
214             size_t cap = capacity();
215             for (size_t i = 0; i != cap; ++i)
216                 slots[i].is_occupied = false;
217             num_occupied_slots = 0;
218         }
219         /*
220             returns true IFF the map contains no elements.
221         */
222         bool is_empty() const {
223             return size() == 0;
224         }
225         /*
226             returns the number of slots in the map.
227         */
228         size_t capacity() const {
229             return curr_capacity;
230         }
231         /*
232             returns the number of items actually stored in the map.
233         */
234         size_t size() const {
235             return num_occupied_slots;
236         }
237         /*
238             returns the map's load factor (size = load * capacity).
239         */
240         double load() const {
241             return static_cast<double>(size()) / capacity();
242         }
243         /*
```

```
244             inserts into the ostream, the backing array's contents in sequential
                    order.
245             Empty slots shall be denoted by a hyphen, non-empty slots by that item's
246             key. [This function will be used for debugging/monitoring].
247         */
248         std::ostream& print(std::ostream& out) const {
249             size_t cap = capacity();
250             out << '[';
251             for (size_t i = 0; i != cap; ++i) {
252                 if (slots[i].is_occupied) {
253                     out << slots[i].item.key.raw();
254                 } else {
255                     out << "-";
256                 }
257                 if (i + 1 < cap)
258                     out << '|';
259             }
260             out << ']';
261             return out;
262         }
263 
264         priority_queue<ClusterInventory> cluster_distribution() {
265             //use an array to count cluster instances, then feed those to a priority
                    queue and return it.
266             priority_queue<ClusterInventory> cluster_pq;
267             if (size() == 0) return cluster_pq;
268             size_t M = capacity();
269             size_t cluster_counter[M + 1];
270             for (size_t i = 0; i <= M; ++i)
271                 cluster_counter[i] = 0;
272             if (size() == M) {
273                 //handle the special case when the map is full
274                 cluster_counter[size()]++;
275             } else {
276                 //have at least one unoccupied slot
277                 bool first_cluster_skipped = false;
278                 size_t curr_cluster_size = 0;
279                 //treat the backing array as a circular buffer and make a maximum of
                        two passes to
280                 //capture everything, including the wraparound cluster if it exists
281                 for (size_t i = 1; i != M * 2; ++i) {
282                     Slot const& curr_slot = slots[i % M], prev_slot = slots[(i - 1) %
                            M];
283                     if (curr_slot.is_occupied && prev_slot.is_occupied) {
284                         //still in a cluster
285                         ++curr_cluster_size;
286                     } else if (curr_slot.is_occupied && prev_slot.is_occupied ==
                            false) {
287                         //found a new cluster
288                         curr_cluster_size = 1;
289                     } else if ( ! curr_slot.is_occupied && prev_slot.is_occupied) {
290                         //found the end of a cluster
```

```
                        if (first_cluster_skipped) {
                            cluster_counter[curr_cluster_size]++;
                            if (i >= M) {
                                //reached the end of the first cluster in the second pass, so no all clusters have been handled
                                break;
                            }
                        } 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 + hash_utils::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();
                    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

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

SSLL(const SSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

SSLL& operator=(const SSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
  - In this case we pass the element to push_back(), which can do O(1) insert
  - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail

- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list

- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

Iterator Methods

explicit SSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Iter(const SSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

Const Iterator Methods

explicit SSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list

- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Const_Iter(const SSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

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_

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

namespace cop3530 {
    double lg(size_t i) {
        return std::log(i) / std::log(2);
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        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;
                }
            };
        };
        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;
        }
        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) {
                    //make capacity a power of 2, greater than the minimum capacity
                    return 1 << static_cast<size_t>(std::ceil(lg(min_capacity)));
                }
            };
```

```
47              struct compare_functor {
48                  int operator()(const char* a, const char* b) const {
49                      int cmp = strcmp(a, b);
50                      return (cmp < 0 ? -1 :
51                                      (cmp > 0 ? 1 : 0));
52                  }
53                  int operator()(double a, double b) const {
54                      return (a < b ? -1 :
55                                      (a > b ? 1 : 0));
56                  }
57                  int operator()(std::string const& a, std::string const& b) const {
58                      return (a < b ? -1 :
59                                      (a > b ? 1 : 0));
60                  }
61                  int operator()(int a, int b) const {
62                      return (a < b ? -1 :
63                                      (a > b ? 1 : 0));
64                  }
65              };
66              namespace primary_hashes {
67                  struct hash_basic {
68                  //this is such a stupid hash method, but unlike my pathetic attempts
                        at implementing
69                  //various other hashing methods, it works and is generalizable to
                        all the required key
70                  //types. together with double hashing it should make for a passable
                        hashing routine.
71                  public:
72                      size_t operator()(const char* key) const {
73                          return str_to_numeric(key);
74                      }
75                      size_t operator()(double key) const {
76                          return static_cast<size_t>(std::fmod(key, max_size_t));
77                      }
78                      size_t operator()(int key) const {
79                          return static_cast<size_t>(key);
80                      }
81                      size_t operator()(std::string const& key) const {
82                          const char* c_key = key.c_str();
83                          return operator()(c_key);
84                      }
85                  };
86              }
87              namespace secondary_hashes {
88                  struct linear_probe {
89                      bool changes_with_probe_attempt() const {
90                          return false;
91                      }
92                      size_t operator()(const char* key, size_t probe_attempt) const {
93                          return 1;
94                      }
95                  };
```

```
96                  struct quadratic_probe {
97                      bool changes_with_probe_attempt() const {
98                          return true;
99                      }
100                     size_t operator()(const char* key, size_t probe_attempt) const {
101                         return probe_attempt;
102                     }
103                 };
104                 struct hash_double {
105                 private:
106                     size_t hash_numeric(size_t numeric) const {
107                         size_t hash = numeric % 97; //simple modulus using a prime
                                number (from algorithms in c++)
108                         //the second hash may not be zero (will cause an infinite
                                loop).
109                         //also, hash must be relatively prime to map_capacity so that
                                every slot can be hit.
110                         //since map capacity is a power of two if we use the capacity
                                planner functor,
111                         //both properties are attainable by adding one to the hash if
                                it is even (despite what my
112                         //7th grade algebra teacher attempted to teach me, I
                                stubbournly consider zero to be an even
113                         //integer despite no formal training in number theory)
114                         bool is_even = (hash & 1) == 0;
115                         if (is_even)
116                             ++hash;
117                         return hash;
118                     }
119                 public:
120                     bool changes_with_probe_attempt() const {
121                         return false;
122                     }
123                     size_t operator()(const char* key, size_t unused) const {
124                         size_t numeric = str_to_numeric(key);
125                         return hash_numeric(numeric);
126                     }
127                     size_t operator()(double key, size_t unused) const {
128                         return hash_numeric(key);
129                     }
130                     size_t operator()(int key, size_t unused) const {
131                         return hash_numeric(key);
132                     }
133                     size_t operator()(std::string key, size_t unused) const {
134                         const char* c_key = key.c_str();
135                         return operator()(c_key, unused);
136                     }
137                 };
138             }
139         }
140     }
141 }
```

```
142
143 std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory
        const& rhs) {
144     out << "Cluster{size=" << rhs.cluster_size << ", instances=" <<
            rhs.num_instances << "}";
145     return out;
146 }
147
148 #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 takes a simple singly linked list containing clusters and exposes
    //a method (get_next_item) which returns the clusters is order of ascending size
    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;
            }
        }
```

```
45      public:
46          //take a linked list of cluster descriptors and add each to the priority
                queue
47          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
48              T empty_item;
49              tree.push_back(empty_item);
50          }
51          priority_queue(priority_queue const& src) {
52              tree = src.tree;
53              num_items = src.num_items;
54          }
55          T get_next_item() {
56              std::swap(tree[1], tree[num_items]);
57              T ret = tree[num_items--];
58              fix_down();
59              return ret;
60          }
61          void add_to_queue(T const& item) {
62              tree.push_back(item);
63              num_items++;
64              fix_up(num_items);
65          }
66          size_t size() {
67              return num_items;
68          }
69          bool empty() {
70              return num_items == 0;
71          }
72      };
73  }
74
75  #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 compare_functor = hash_utils::functors::compare_functor,
             typename primary_hash =
                 hash_utils::functors::primary_hashes::hash_basic,
             typename secondary_hash =
                 hash_utils::functors::secondary_hashes::hash_double>
    class HashMapBucketsGeneric {
    private:
        typedef hash_utils::ClusterInventory ClusterInventory;
        class Key {
        private:
            key_type raw_key;
            compare_functor compare;
            primary_hash hasher1;
            secondary_hash hasher2;
            size_t hash1_val;
            size_t hash2_val;
            size_t old_map_capacity;
        public:
            bool operator==(Key const& rhs) const {
                return compare(raw_key, rhs.raw_key) == 0;
            }
            bool operator==(key_type const& rhs) const {
                return compare(raw_key, rhs) == 0;
            }
            bool operator!=(Key const& rhs) const {
                return ! operator==(rhs);
            }
            bool operator!=(key_type const& rhs) const {
                return ! operator==(rhs);
            }
            size_t hash(size_t map_capacity, size_t probe_attempt) const {
                size_t local_hash2_val;
                if (probe_attempt != 0 && hasher2.changes_with_probe_attempt())
                {
```

```
                //if the hashing function value is dependent on the probe attempt
                //(eg quadratic probing), then we need to retrieve the new value
                local_hash2_val = hasher2(raw_key, probe_attempt);
            } else {
                //otherwise we can just use the value we have stored
                local_hash2_val = hash2_val;
            }
            return (hash1_val + probe_attempt * local_hash2_val) % map_capacity;
        }
        key_type const& raw() const {
            return raw_key;
        }
        void reset(key_type const& key) {
            raw_key = key;
            size_t base_probe_attempt = 0;
            hash1_val = hasher1(key);
            hash2_val = hasher2(key, base_probe_attempt);
        }
        explicit Key(key_type key) {
            reset(key);
        }
        Key() = default;
    };
    class Value {
    private:
        value_type raw_value;
    public:
        bool operator==(Value const& rhs) const {
            return compare(raw_value, rhs.raw_value);
        }
        bool operator==(value_type const& rhs) const {
            return compare(raw_value, rhs) == 0;
        }
        value_type const& raw() const {
            return raw_value;
        }
        explicit Value(value_type value): raw_value(value) {}
        Value() = default;
    };
    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;
```

```
96              }
97              ~Bucket() {
98                  while ( ! head->is_dummy) {
99                      Item* to_delete = head;
100                     head = head->next;
101                     delete to_delete;
102                 }
103                 delete head; //tail
104             }
105         };
106         typedef Item* link;
107         Bucket* buckets;
108         size_t num_buckets = 0;
109         size_t num_items = 0;
110         /*
111             searches the bucket corresponding to the specified key's hash for that
112             key. if found, stores a reference to that item and returns P, the number
                    of
113             probe attempts needed to get to the item (ie the number of chain links
                    needed
114             to be traversed). otherwise return -1 * P and stores the pointer to the
                    tail dummy node in
115             item_ptr.
116         */
117         int search_internal(Key const& key, link& item_ptr) {
118             int probe_attempts = 1;
119             size_t hash_val = key.hash(capacity(), 0);
120             Bucket& bucket = buckets[hash_val];
121             item_ptr = bucket.head;
122             while ( ! item_ptr->is_dummy) {
123                 if (item_ptr->key == key) {
124                     //found the key
125                     return probe_attempts;
126                 }
127                 item_ptr = item_ptr->next;
128                 ++probe_attempts;
129             }
130             //key not found
131             return probe_attempts * -1;
132         }
133         void init() {
134             buckets = new Bucket[num_buckets];
135             num_items = 0;
136         }
137     public:
138         HashMapBucketsGeneric(size_t const min_buckets)
139         {
140             if (min_buckets == 0) {
141                 throw std::domain_error("min_buckets must be at least 1");
142             }
143             cop3530::hash_utils::functors::map_capacity_planner capacity_planner;
```

```
144             num_buckets = capacity_planner(min_buckets); //make capacity a power of
                    2, greater than the minimum capacity
145             init();
146         }
147         ~HashMapBucketsGeneric() {
148             delete[] buckets;
149         }
150         /*
151             if there is space available, adds the specified key/value-pair to the
                    hash map and returns the
152             number of probes required, P; otherwise returns -1 * P (that's a lie: we
                    will always have space
153             available because each bucket contains a linked list that is
                    indefinitely growable). If an item
154             already exists in the map with the same key, replace its value.
155         */
156         int insert(key_type const& key, value_type const& value) {
157             Item* item;
158             Key k(key);
159             Value v(value);
160             int probes_required = search_internal(k, item);
161             if (probes_required > 0)
162                 //found item
163                 item->value = v;
164             else {
165                 //currently holding tail (item not found). transform it into a valid
                        item then add a new tail
166                 item->is_dummy = false;
167                 item->key = k;
168                 item->value = v;
169                 item->next = new Item(nullptr);
170                 ++num_items;
171             }
172             return std::abs(probes_required);
173         }
174         /*
175             if there is an item matching key, removes the key/value-pair from the
                    map, stores it's value in
176             value, and returns the number of probes required, P; otherwise returns
                    -1 * P.
177         */
178         int remove(key_type const& key, value_type& value) {
179             Key k(key);
180             Item* item;
181             int probes_required = search_internal(key, item);
182             if (probes_required > 0) {
183                 //found item
184                 value = item->value.raw();
185                 //swap the current item for the next one
186                 Item* to_delete = item->next;
187                 *item = *to_delete;
188                 delete to_delete;
```

```
189                 --num_items;
190             }
191             return probes_required;
192         }
193         /*
194             if there is an item matching key, stores it's value in value, and
                returns the
195             number of probes required, P; otherwise returns -1 * P. Regardless, the
                item
196             remains in the map.
197         */
198         int search(key_type const& key, value_type& value) {
199             Item* item;
200             Key k(key);
201             int probes_required = search_internal(k, item);
202             if (probes_required > 0) {
203                 //found item
204                 value = item->value.raw();
205             }
206             return probes_required;
207         }
208         /*
209             removes all items from the map.
210         */
211         void clear() {
212             delete buckets;
213             init();
214         }
215         /*
216             returns true IFF the map contains no elements.
217         */
218         bool is_empty() {
219             return size() == 0;
220         }
221         /*
222             returns the number of slots in the map.
223         */
224         size_t capacity() {
225             return num_buckets;
226         }
227         /*
228             returns the number of items actually stored in the map.
229         */
230         size_t size() {
231             return num_items;
232         }
233         /*
234             returns the map's load factor (occupied buckets = load * capacity).
235         */
236         double load() {
237             size_t occupied_buckets = 0;
238             if (size() > 0) {
```

```
239             size_t M = capacity();
240             for (size_t i = 0; i != M; ++i) {
241                 Bucket const& bucket = buckets[i];
242                 if ( ! bucket.head->is_dummy)
243                     //bucket has at least one item
244                     occupied_buckets++;
245             }
246         }
247         return static_cast<double>(occupied_buckets) / capacity();
248     }
249     /*
250         inserts into the ostream, the backing array's contents in sequential
                order.
251         Empty slots shall be denoted by a hyphen, non-empty slots by that item's
252         key. [This function will be used for debugging/monitoring].
253     */
254     std::ostream& print(std::ostream& out) {
255         size_t cap = capacity();
256         bool print_separator = false;
257         out << '[';
258         for (size_t i = 0; i != cap; ++i) {
259             Bucket const& bucket = buckets[i];
260             for (Item* item = bucket.head; item->is_dummy != true; item =
                    item->next) {
261                 if (print_separator)
262                     out << "|";
263                 else
264                     print_separator = true;
265                 out << item->key.raw();
266             }
267         }
268         out << ']';
269         return out;
270     }
271
272     /*
273         returns a priority queue containing cluster sizes and instances (in the
                form of ClusterInventory
274         struct instances), sorted by cluster size.
275     */
276     priority_queue<ClusterInventory> cluster_distribution() {
277         //use a simple linked list to count cluster instances, then feed those
                to a priority queue and return it.
278         priority_queue<ClusterInventory> cluster_pq;
279         if (size() == 0) return cluster_pq;
280         SSLL<ClusterInventory> clusters;
281         size_t M = capacity();
282         for (size_t i = 0; i != M; ++i) {
283             Bucket const& bucket = buckets[i];
284             size_t bucket_size = 0;
285             Item* item_ptr = bucket.head;
286             while ( ! item_ptr->is_dummy) {
```

```
287                 ++bucket_size;
288                 item_ptr = item_ptr->next;
289             }
290             //I don't love this O(N^2) implementation, but premature
                    optimization is the root of all evil and late projects
291             SSLL<ClusterInventory>::iterator cluster_iterator = clusters.begin();
292             SSLL<ClusterInventory>::iterator cluster_iterator_end =
                    clusters.end();
293             bool found_cluster = false;
294             for (; cluster_iterator != cluster_iterator_end; ++cluster_iterator)
                    {
295                 if (cluster_iterator->cluster_size == bucket_size) {
296                     found_cluster = true;
297                     break;
298                 }
299             }
300             if (found_cluster)
301                 cluster_iterator->num_instances++;
302             else
303                 clusters.push_back({bucket_size, 1});
304         }
305         SSLL<ClusterInventory>::const_iterator cluster_iterator =
                clusters.begin();
306         SSLL<ClusterInventory>::const_iterator cluster_iterator_end =
                clusters.end();
307         for (; cluster_iterator != cluster_iterator_end; ++cluster_iterator) {
308             if (cluster_iterator->cluster_size > 0)
309                 cluster_pq.add_to_queue(*cluster_iterator);
310         }
311         return cluster_pq;
312     }
313
314     /*
315         generate a random number, R, (1,size), and starting with slot zero in
                the backing array,
316         find the R-th occupied slot; remove the item from that slot (adjusting
                subsequent items as
317         necessary), and return its key.
318     */
319     key_type remove_random() {
320         if (size() == 0) throw std::logic_error("Cant remove from an empty map");
321         size_t num_slots = capacity();
322         size_t ith_node_to_delete = 1 + hash_utils::rand_i(size());
323         for (size_t i = 0; i != num_slots; ++i) {
324             Bucket const& bucket = buckets[i];
325             Item* item_ptr = bucket.head;
326             while ( ! item_ptr->is_dummy) {
327                 if (--ith_node_to_delete == 0) {
328                     key_type key = item_ptr->key.raw();
329                     value_type val_dummy;
330                     remove(key, val_dummy);
331                     return key;
```

```
                    }
                    item_ptr = item_ptr->next;
                }
            }
            throw std::logic_error("Unexpected end of remove_random function");
        }
    };
}

#endif
```

Part IV: Randomized BST

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

SSLL(const SSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

SSLL& operator=(const SSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
  - In this case we pass the element to push_back(), which can do O(1) insert
  - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail

- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list

- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

Iterator Methods

explicit SSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Iter(const SSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

Const Iterator Methods

explicit SSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list

- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Const_Iter(const SSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

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
* 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_

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

namespace cop3530 {
    double lg(size_t i) {
        return std::log(i) / std::log(2);
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        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;
                }
            };
        };
        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;
        }
        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) {
                    //make capacity a power of 2, greater than the minimum capacity
                    return 1 << static_cast<size_t>(std::ceil(lg(min_capacity)));
                }
            };
```

```
47              struct compare_functor {
48                  int operator()(const char* a, const char* b) const {
49                      int cmp = strcmp(a, b);
50                      return (cmp < 0 ? -1 :
51                                      (cmp > 0 ? 1 : 0));
52                  }
53                  int operator()(double a, double b) const {
54                      return (a < b ? -1 :
55                                      (a > b ? 1 : 0));
56                  }
57                  int operator()(std::string const& a, std::string const& b) const {
58                      return (a < b ? -1 :
59                                      (a > b ? 1 : 0));
60                  }
61                  int operator()(int a, int b) const {
62                      return (a < b ? -1 :
63                                      (a > b ? 1 : 0));
64                  }
65              };
66              namespace primary_hashes {
67                  struct hash_basic {
68                  //this is such a stupid hash method, but unlike my pathetic attempts
                        at implementing
69                  //various other hashing methods, it works and is generalizable to
                        all the required key
70                  //types. together with double hashing it should make for a passable
                        hashing routine.
71                  public:
72                      size_t operator()(const char* key) const {
73                          return str_to_numeric(key);
74                      }
75                      size_t operator()(double key) const {
76                          return static_cast<size_t>(std::fmod(key, max_size_t));
77                      }
78                      size_t operator()(int key) const {
79                          return static_cast<size_t>(key);
80                      }
81                      size_t operator()(std::string const& key) const {
82                          const char* c_key = key.c_str();
83                          return operator()(c_key);
84                      }
85                  };
86              }
87              namespace secondary_hashes {
88                  struct linear_probe {
89                      bool changes_with_probe_attempt() const {
90                          return false;
91                      }
92                      size_t operator()(const char* key, size_t probe_attempt) const {
93                          return 1;
94                      }
95                  };
```

```
96                  struct quadratic_probe {
97                      bool changes_with_probe_attempt() const {
98                          return true;
99                      }
100                     size_t operator()(const char* key, size_t probe_attempt) const {
101                         return probe_attempt;
102                     }
103                 };
104                 struct hash_double {
105                 private:
106                     size_t hash_numeric(size_t numeric) const {
107                         size_t hash = numeric % 97; //simple modulus using a prime
                                number (from algorithms in c++)
108                         //the second hash may not be zero (will cause an infinite
                                loop).
109                         //also, hash must be relatively prime to map_capacity so that
                                every slot can be hit.
110                         //since map capacity is a power of two if we use the capacity
                                planner functor,
111                         //both properties are attainable by adding one to the hash if
                                it is even (despite what my
112                         //7th grade algebra teacher attempted to teach me, I
                                stubbournly consider zero to be an even
113                         //integer despite no formal training in number theory)
114                         bool is_even = (hash & 1) == 0;
115                         if (is_even)
116                             ++hash;
117                         return hash;
118                     }
119                 public:
120                     bool changes_with_probe_attempt() const {
121                         return false;
122                     }
123                     size_t operator()(const char* key, size_t unused) const {
124                         size_t numeric = str_to_numeric(key);
125                         return hash_numeric(numeric);
126                     }
127                     size_t operator()(double key, size_t unused) const {
128                         return hash_numeric(key);
129                     }
130                     size_t operator()(int key, size_t unused) const {
131                         return hash_numeric(key);
132                     }
133                     size_t operator()(std::string key, size_t unused) const {
134                         const char* c_key = key.c_str();
135                         return operator()(c_key, unused);
136                     }
137                 };
138             }
139         }
140     }
141 }
```

```
142
143 std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory
        const& rhs) {
144     out << "Cluster{size=" << rhs.cluster_size << ", instances=" <<
            rhs.num_instances << "}";
145     return out;
146 }
147
148 #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 takes a simple singly linked list containing clusters and exposes
    //a method (get_next_item) which returns the clusters is order of ascending size
    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;
            }
        }
```

```
45      public:
46          //take a linked list of cluster descriptors and add each to the priority
                queue
47          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
48              T empty_item;
49              tree.push_back(empty_item);
50          }
51          priority_queue(priority_queue const& src) {
52              tree = src.tree;
53              num_items = src.num_items;
54          }
55          T get_next_item() {
56              std::swap(tree[1], tree[num_items]);
57              T ret = tree[num_items--];
58              fix_down();
59              return ret;
60          }
61          void add_to_queue(T const& item) {
62              tree.push_back(item);
63              num_items++;
64              fix_up(num_items);
65          }
66          size_t size() {
67              return num_items;
68          }
69          bool empty() {
70              return num_items == 0;
71          }
72      };
73  }
74
75  #endif // _PRIORITY_QUEUE_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,
             typename compare_functor = hash_utils::functors::compare_functor>
    class BST {
    protected: //let RBST and AVL inherit everything
        typedef hash_utils::ClusterInventory ClusterInventory;
        compare_functor compare;
        struct Node;
        typedef Node* link;
        struct Node {
            key_type key;
            value_type 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 get_height_recursive(Node* nodes) {
                //this function is for debugging purposes, does recursive traversal
                    to find the correct height
                //todo: delete this function
                size_t left_height = 0, right_height = 0;
                size_t calculated_height = 0;
                if (left_index)
                    left_height = nodes[left_index].get_height_recursive(nodes);
                if (right_index)
                    right_height = nodes[right_index].get_height_recursive(nodes);
                calculated_height = 1 + std::max(left_height, right_height);
                return calculated_height;
            }
            void update_height(Node* nodes) {
                //note: this method depends on the left and right subtree heights
                    being correct
                size_t left_height = 0, right_height = 0;
                if (left_index)
                    left_height = nodes[left_index].height;
```

```
45              if (right_index)
46                  right_height = nodes[right_index].height;
47              height = 1 + std::max(left_height, right_height);
48              //todo: delete the following expensive check, or move it into DEBUG
                    condition
49              size_t calculated_height = get_height_recursive(nodes);
50              if (calculated_height != height) {
51                  std::ostringstream msg;
52                  msg << "Manually calculated height, " << calculated_height << ",
                        different than tracked height, " << height;
53                  throw std::runtime_error(msg.str());
54              }
55          }
56          void disable_and_adopt_free_tree(size_t free_index) {
57              is_occupied = false;
58              height = 0;
59              num_children = 0;
60              right_index = 0;
61              left_index = free_index;
62          }
63          void reset_and_enable(key_type const new_key, value_type const&
                new_value) {
64              is_occupied = true;
65              height = 1; //self
66              left_index = right_index = 0;
67              num_children = 0;
68              key = new_key;
69              value = new_value;
70          }
71          int balance_factor(const Node* nodes) const {
72              size_t left_height = 0, right_height = 0;
73              if (left_index)
74                  left_height = nodes[left_index].height;
75              if (right_index)
76                  right_height = nodes[right_index].height;
77              return static_cast<long int>(left_height) - static_cast<long
                    int>(right_height);
78          }
79      };
80      Node* nodes; //***note: array is 1-based so leaf nodes have child indices
            set to zero
81      size_t free_index;
82      size_t root_index;
83      size_t curr_capacity;
84      virtual size_t remove_smallest_key_node_index(size_t& subtree_root_index) {
85          //returns the index of the node with the smallest key, while
86          //setting its parent's left child index to the smallest key node's
87          //right child index. recursion downward through this function updates
88          //the heights of the nodes it traverses
89          Node& subtree_root = nodes[subtree_root_index];
90          size_t smallest_key_node_index = 0;
91          if (subtree_root_index == 0) {
```

```
92                throw std::logic_error("Expected to find a valid node, but didn't");
93            } else {
94                if (subtree_root.left_index) {
95                    smallest_key_node_index =
                          remove_smallest_key_node_index(subtree_root.left_index);
96                    subtree_root.num_children--;
97                    subtree_root.update_height(nodes);
98                } else {
99                    smallest_key_node_index = subtree_root_index;
100                   subtree_root_index = subtree_root.right_index;
101               }
102           }
103           return smallest_key_node_index;
104       }
105       virtual size_t remove_largest_key_node_index(size_t& subtree_root_index) {
106           //returns the index of the node with the largest key, while
107           //setting its parent's right child index to the largest key node's
108           //left child index. recursion downward through this function updates
109           //the heights of the nodes it traverses
110           Node& subtree_root = nodes[subtree_root_index];
111           size_t largest_key_node_index = 0;
112           if (subtree_root_index == 0) {
113               throw std::logic_error("Expected to find a valid node, but didn't");
114           } else {
115               if (subtree_root.right_index) {
116                   largest_key_node_index =
                          remove_largest_key_node_index(subtree_root.right_index);
117                   subtree_root.num_children--;
118                   subtree_root.update_height(nodes);
119               } else {
120                   largest_key_node_index = subtree_root_index;
121                   subtree_root_index = subtree_root.left_index;
122               }
123           }
124           return largest_key_node_index;
125       }
126       virtual void remove_node(size_t& subtree_root_index) {
127           Node& subtree_root = nodes[subtree_root_index];
128           size_t index_to_delete = subtree_root_index;
129           if (subtree_root.right_index || subtree_root.left_index) {
130               //subtree has at least one child
131               if (subtree_root.right_index)
132                   //replace the root with the smallest-keyed node in the right
                          subtree
133                   subtree_root_index =
                          remove_smallest_key_node_index(subtree_root.right_index);
134               else if (subtree_root.left_index)
135                   //replace the root with the largest-keyed node in the left subtree
136                   subtree_root_index =
                          remove_largest_key_node_index(subtree_root.left_index);
137               //have the new root adopt the old root's children
138               Node& new_root = nodes[subtree_root_index];
```

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

```
                    subtree_root.update_height(nodes);
                }
                break;
            case 0:
                //found key, remove the node
                found_key = true;
                value = subtree_root.value;
                remove_node(subtree_root_index);
                break;
            default:
                throw std::domain_error("Unexpected compare() function return
                    value");
            }
        }
        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 << "]";
        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--] += " ";
            write_subtree_buffer(subtree_root.right_index,
                                 buffer_lines,
                                 right_child_line_index,
                                 lbound_line_index,
                                 root_line_index);
        }
        //print the left descendents
```

```
235             if (subtree_root.left_index > 0) {
236                 //at least 1 left child
237                 size_t bottom_dashes = 1;
238                 Node const& left_child = nodes[subtree_root.left_index];
239                 if (left_child.right_index > 0) {
240                     //left child has at least 1 right child
241                     Node const& left_right_child = nodes[left_child.right_index];
242                     bottom_dashes += 2 * (1 + left_right_child.num_children);
243                 }
244                 size_t bottom_line_index = root_line_index + 1;
245                 while (bottom_line_index <= root_line_index + bottom_dashes)
246                     buffer_lines[bottom_line_index++] += "| ";
247                 size_t left_child_line_index = bottom_line_index;
248                 buffer_lines[bottom_line_index++] += "+--";
249                 while (bottom_line_index < ubound_line_index)
250                     buffer_lines[bottom_line_index++] += " ";
251                 write_subtree_buffer(subtree_root.left_index,
252                                      buffer_lines,
253                                      left_child_line_index,
254                                      root_line_index + 1,
255                                      ubound_line_index);
256             }
257         }
258         void add_node_to_free_tree(size_t node_index) {
259             nodes[node_index].disable_and_adopt_free_tree(free_index);
260             free_index = node_index;
261         }
262         size_t procure_node(key_type const& key, value_type const& value) {
263             //updates the free index to the first free node's left child (while
                    transforming that first free
264             //node to an enabled node with the specified key/value) and returns the
                    index of what was the last
265             //free index
266             size_t node_index = free_index;
267             free_index = nodes[free_index].left_index;
268             Node& n = nodes[node_index];
269             n.reset_and_enable(key, value);
270             return node_index;
271         }
272         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)
273                                    size_t& subtree_root_index,
274                                    key_type const& key,
275                                    value_type const& value,
276                                    bool& found_key)
277         {
278             if (subtree_root_index == 0) {
279                 //key not found
280                 subtree_root_index = procure_node(key, value);
281             } else {
282                 //parent was not a leaf
283                 //keep going down to the base of the tree
```

```
284             Node& subtree_root = nodes[subtree_root_index];
285             ++nodes_visited;
286             switch (compare(key, subtree_root.key)) {
287             case -1:
288                 //key is less than subtree root's key
289                 nodes_visited = insert_at_leaf(nodes_visited,
                        subtree_root.left_index, key, value, found_key);
290                 if ( ! found_key) {
291                     //given key is unique to the tree, so a new node was added
292                     subtree_root.num_children++;
293                     subtree_root.update_height(nodes);
294                 }
295                 break;
296             case 1:
297                 //key is greater than subtree root's key
298                 nodes_visited = insert_at_leaf(nodes_visited,
                        subtree_root.right_index, key, value, found_key);
299                 if ( ! found_key) {
300                     //given key is unique to the tree, so a new node was added
301                     subtree_root.num_children++;
302                     subtree_root.update_height(nodes);
303                 }
304                 break;
305             case 0:
306                 //found key, replace the value
307                 subtree_root.value = value;
308                 found_key = true;
309                 break;
310             default:
311                 throw std::domain_error("Unexpected compare() function return
                        value");
312             }
313         }
314         return nodes_visited;
315     }
316     void rotate_left(size_t& subtree_root_index) {
317         Node& subtree_root = nodes[subtree_root_index];
318         size_t right_child_index = subtree_root.right_index;
319         Node& right_child = nodes[right_child_index];
320
321         //original root adopts the right child's left subtree
322         subtree_root.right_index = right_child.left_index;
323         //original root adopted a subtree (whose height did not change), so
                update its height
324         subtree_root.update_height(nodes);
325
326         //right child adopts original root and its children
327         right_child.left_index = subtree_root_index;
328         //right child (new root) adopted the original root (whose height has
                been updated), so update its height
329         right_child.update_height(nodes);
```

```
330             //since right child took the subtree root's place, it has the same
                    number of children as the original root
331             right_child.num_children = subtree_root.num_children;
332
333             //root has new children, so update that counter (done after changing the
                    right child's children counter
334             //because that depends on the original root's counter)
335             subtree_root.num_children = 0;
336             if (subtree_root.left_index != 0)
337                 subtree_root.num_children += 1 +
                        nodes[subtree_root.left_index].num_children;
338             if (subtree_root.right_index != 0)
339                 subtree_root.num_children += 1 +
                        nodes[subtree_root.right_index].num_children;
340
341             //set the right child as the new root
342             subtree_root_index = right_child_index;
343         }
344         void rotate_right(size_t& subtree_root_index) {
345             Node& subtree_root = nodes[subtree_root_index];
346             size_t left_child_index = subtree_root.left_index;
347             Node& left_child = nodes[left_child_index];
348
349             //original root adopts the left child's right subtree
350             subtree_root.left_index = left_child.right_index;
351             //original root adopted a subtree (whose height did not change), so
                    update its height
352             subtree_root.update_height(nodes);
353
354             //left child adopts original root and its children
355             left_child.right_index = subtree_root_index;
356             //left child (new root) adopted the original root (whose height has been
                    updated), so update its height
357             left_child.update_height(nodes);
358             //since left child took the subtree root's place, it has the same number
                    of children as the original root
359             left_child.num_children = subtree_root.num_children;
360
361             //root has new children, so update that counter (done after changing the
                    left child's children counter
362             //because that depends on the original root's counter)
363             subtree_root.num_children = 0;
364             if (subtree_root.left_index != 0)
365                 subtree_root.num_children += 1 +
                        nodes[subtree_root.left_index].num_children;
366             if (subtree_root.right_index != 0)
367                 subtree_root.num_children += 1 +
                        nodes[subtree_root.right_index].num_children;
368
369             //set the left child as the new root
370             subtree_root_index = left_child_index;
371         }
```

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

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

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

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

```
560             returns a list indicating the number of leaf nodes at each height (since
                    the RBST doesn't exhibit
561             true clustering, but can have degenerate branches).
562         */
563         virtual priority_queue<hash_utils::ClusterInventory> cluster_distribution()
                {
564             //use an array to count cluster instances, then feed those to a priority
                    queue and return it.
565             priority_queue<ClusterInventory> cluster_pq;
566             if (is_empty()) return cluster_pq;
567             size_t max_height = nodes[root_index].height;
568             size_t cluster_counter[max_height + 1];
569             for (size_t i = 0; i <= max_height; ++i)
570                 cluster_counter[i] = 0;
571             prepare_cluster_distribution(root_index, 1, cluster_counter);
572             for (size_t i = 1; i <= max_height; ++i)
573                 if (cluster_counter[i] > 0) {
574                     ClusterInventory cluster{i, cluster_counter[i]};
575                     cluster_pq.add_to_queue(cluster);
576                 }
577             return cluster_pq;
578         }
579 
580         /*
581             generate a random number, R, (1,size), and starting with the root (node
                    1), do an in-order
582             traversal to find the R-th occupied node; remove that node (adjusting
                    its children accordingly),
583             and return its key.
584         */
585         virtual key_type remove_random() {
586             if (size() == 0) throw std::logic_error("Cant remove from an empty map");
587             size_t ith_node_to_delete = 1 + hash_utils::rand_i(size());
588             key_type key;
589             remove_ith_node_inorder(root_index, ith_node_to_delete, key);
590             return key;
591         }
592     };
593 }
594 
595 #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,
             typename compare_functor = hash_utils::functors::compare_functor>
    class RBST: public BST<key_type, value_type, compare_functor> {
    /*
        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, compare_functor>;
        using typename super::Node;
        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_type const& key,
                           value_type 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,
                           size_t& subtree_root_index,
```

```
43                      key_type const& key,
44                      value_type const& value,
45                      bool& found_key)
46      {
47          if (subtree_root_index == 0) {
48              //parent was a leaf, so create a new leaf
49              subtree_root_index = this->procure_node(key, value);
50          } else {
51              //parent was not a leaf
52              Node& subtree_root = this->nodes[subtree_root_index];
53              ++nodes_visited;
54              //keep going down to the base of the tree
55              switch (this->compare(key, subtree_root.key)) {
56              case -1:
57                  //key is less than subtree root's 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                      //current subtree root may have had its height changed, so
                            update that before
66                      //promoting the new node
67                      subtree_root.update_height(this->nodes);
68                      this->rotate_right(subtree_root_index);
69                  }
70                  break;
71              case 1:
72                  //key is greater than subtree root's key
73                  nodes_visited = insert_at_root(nodes_visited,
                        subtree_root.right_index, key, value, found_key);
74                  if ( ! found_key) {
75                      subtree_root.num_children++;
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                  break;
82              case 0:
83                  //found key, replace the value
84                  subtree_root.value = value;
85                  found_key = true;
86                  break;
87              default:
```

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

Part IV BONUS: AVL Tree

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

SSLL(const SSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

SSLL& operator=(const SSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
  - In this case we pass the element to push_back(), which can do O(1) insert
  - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail

- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list

- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

Iterator Methods

explicit SSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Iter(const SSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

Const Iterator Methods

explicit SSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list

- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

SSLL_Const_Iter(const SSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

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 part5/source
./compile.sh
./run_tests > output.txt
```

common/common.h

common/common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

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

namespace cop3530 {
    double lg(size_t i) {
        return std::log(i) / std::log(2);
    }

    namespace hash_utils {
        static constexpr size_t max_size_t = std::numeric_limits<size_t>::max();
        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;
                }
            };
        };
        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;
        }
        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) {
                    //make capacity a power of 2, greater than the minimum capacity
                    return 1 << static_cast<size_t>(std::ceil(lg(min_capacity)));
                }
            };
```

```
47              struct compare_functor {
48                  int operator()(const char* a, const char* b) const {
49                      int cmp = strcmp(a, b);
50                      return (cmp < 0 ? -1 :
51                                      (cmp > 0 ? 1 : 0));
52                  }
53                  int operator()(double a, double b) const {
54                      return (a < b ? -1 :
55                                      (a > b ? 1 : 0));
56                  }
57                  int operator()(std::string const& a, std::string const& b) const {
58                      return (a < b ? -1 :
59                                      (a > b ? 1 : 0));
60                  }
61                  int operator()(int a, int b) const {
62                      return (a < b ? -1 :
63                                      (a > b ? 1 : 0));
64                  }
65              };
66              namespace primary_hashes {
67                  struct hash_basic {
68                  //this is such a stupid hash method, but unlike my pathetic attempts
                        at implementing
69                  //various other hashing methods, it works and is generalizable to
                        all the required key
70                  //types. together with double hashing it should make for a passable
                        hashing routine.
71                  public:
72                      size_t operator()(const char* key) const {
73                          return str_to_numeric(key);
74                      }
75                      size_t operator()(double key) const {
76                          return static_cast<size_t>(std::fmod(key, max_size_t));
77                      }
78                      size_t operator()(int key) const {
79                          return static_cast<size_t>(key);
80                      }
81                      size_t operator()(std::string const& key) const {
82                          const char* c_key = key.c_str();
83                          return operator()(c_key);
84                      }
85                  };
86              }
87              namespace secondary_hashes {
88                  struct linear_probe {
89                      bool changes_with_probe_attempt() const {
90                          return false;
91                      }
92                      size_t operator()(const char* key, size_t probe_attempt) const {
93                          return 1;
94                      }
95                  };
```

```
96                 struct quadratic_probe {
97                     bool changes_with_probe_attempt() const {
98                         return true;
99                     }
100                    size_t operator()(const char* key, size_t probe_attempt) const {
101                        return probe_attempt;
102                    }
103                };
104                struct hash_double {
105                private:
106                    size_t hash_numeric(size_t numeric) const {
107                        size_t hash = numeric % 97; //simple modulus using a prime
                               number (from algorithms in c++)
108                        //the second hash may not be zero (will cause an infinite
                               loop).
109                        //also, hash must be relatively prime to map_capacity so that
                               every slot can be hit.
110                        //since map capacity is a power of two if we use the capacity
                               planner functor,
111                        //both properties are attainable by adding one to the hash if
                               it is even (despite what my
112                        //7th grade algebra teacher attempted to teach me, I
                               stubbournly consider zero to be an even
113                        //integer despite no formal training in number theory)
114                        bool is_even = (hash & 1) == 0;
115                        if (is_even)
116                            ++hash;
117                        return hash;
118                    }
119                public:
120                    bool changes_with_probe_attempt() const {
121                        return false;
122                    }
123                    size_t operator()(const char* key, size_t unused) const {
124                        size_t numeric = str_to_numeric(key);
125                        return hash_numeric(numeric);
126                    }
127                    size_t operator()(double key, size_t unused) const {
128                        return hash_numeric(key);
129                    }
130                    size_t operator()(int key, size_t unused) const {
131                        return hash_numeric(key);
132                    }
133                    size_t operator()(std::string key, size_t unused) const {
134                        const char* c_key = key.c_str();
135                        return operator()(c_key, unused);
136                    }
137                };
138            }
139        }
140    }
141 }
```

```
142
143 std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory
        const& rhs) {
144     out << "Cluster{size=" << rhs.cluster_size << ", instances=" <<
            rhs.num_instances << "}";
145     return out;
146 }
147
148 #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 takes a simple singly linked list containing clusters and exposes
    //a method (get_next_item) which returns the clusters is order of ascending size
    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;
            }
        }
```

```
45      public:
46          //take a linked list of cluster descriptors and add each to the priority
                queue
47          priority_queue(size_t preallocation_size = 100): tree(preallocation_size +
                1) {
48              T empty_item;
49              tree.push_back(empty_item);
50          }
51          priority_queue(priority_queue const& src) {
52              tree = src.tree;
53              num_items = src.num_items;
54          }
55          T get_next_item() {
56              std::swap(tree[1], tree[num_items]);
57              T ret = tree[num_items--];
58              fix_down();
59              return ret;
60          }
61          void add_to_queue(T const& item) {
62              tree.push_back(item);
63              num_items++;
64              fix_up(num_items);
65          }
66          size_t size() {
67              return num_items;
68          }
69          bool empty() {
70              return num_items == 0;
71          }
72      };
73  }
74
75  #endif // _PRIORITY_QUEUE_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,
             typename compare_functor = hash_utils::functors::compare_functor>
    class BST {
    protected: //let RBST and AVL inherit everything
        typedef hash_utils::ClusterInventory ClusterInventory;
        compare_functor compare;
        struct Node;
        typedef Node* link;
        struct Node {
            key_type key;
            value_type 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 get_height_recursive(Node* nodes) {
                //this function is for debugging purposes, does recursive traversal
                    to find the correct height
                //todo: delete this function
                size_t left_height = 0, right_height = 0;
                size_t calculated_height = 0;
                if (left_index)
                    left_height = nodes[left_index].get_height_recursive(nodes);
                if (right_index)
                    right_height = nodes[right_index].get_height_recursive(nodes);
                calculated_height = 1 + std::max(left_height, right_height);
                return calculated_height;
            }
            void update_height(Node* nodes) {
                //note: this method depends on the left and right subtree heights
                    being correct
                size_t left_height = 0, right_height = 0;
                if (left_index)
                    left_height = nodes[left_index].height;
```

```
45              if (right_index)
46                  right_height = nodes[right_index].height;
47              height = 1 + std::max(left_height, right_height);
48              //todo: delete the following expensive check, or move it into DEBUG
                    condition
49              size_t calculated_height = get_height_recursive(nodes);
50              if (calculated_height != height) {
51                  std::ostringstream msg;
52                  msg << "Manually calculated height, " << calculated_height << ",
                        different than tracked height, " << height;
53                  throw std::runtime_error(msg.str());
54              }
55          }
56          void disable_and_adopt_free_tree(size_t free_index) {
57              is_occupied = false;
58              height = 0;
59              num_children = 0;
60              right_index = 0;
61              left_index = free_index;
62          }
63          void reset_and_enable(key_type const new_key, value_type const&
                new_value) {
64              is_occupied = true;
65              height = 1; //self
66              left_index = right_index = 0;
67              num_children = 0;
68              key = new_key;
69              value = new_value;
70          }
71          int balance_factor(const Node* nodes) const {
72              size_t left_height = 0, right_height = 0;
73              if (left_index)
74                  left_height = nodes[left_index].height;
75              if (right_index)
76                  right_height = nodes[right_index].height;
77              return static_cast<long int>(left_height) - static_cast<long
                    int>(right_height);
78          }
79      };
80      Node* nodes; //***note: array is 1-based so leaf nodes have child indices
            set to zero
81      size_t free_index;
82      size_t root_index;
83      size_t curr_capacity;
84      virtual size_t remove_smallest_key_node_index(size_t& subtree_root_index) {
85          //returns the index of the node with the smallest key, while
86          //setting its parent's left child index to the smallest key node's
87          //right child index. recursion downward through this function updates
88          //the heights of the nodes it traverses
89          Node& subtree_root = nodes[subtree_root_index];
90          size_t smallest_key_node_index = 0;
91          if (subtree_root_index == 0) {
```

```
92                throw std::logic_error("Expected to find a valid node, but didn't");
93            } else {
94                if (subtree_root.left_index) {
95                    smallest_key_node_index =
                          remove_smallest_key_node_index(subtree_root.left_index);
96                    subtree_root.num_children--;
97                    subtree_root.update_height(nodes);
98                } else {
99                    smallest_key_node_index = subtree_root_index;
100                   subtree_root_index = subtree_root.right_index;
101               }
102           }
103           return smallest_key_node_index;
104       }
105       virtual size_t remove_largest_key_node_index(size_t& subtree_root_index) {
106           //returns the index of the node with the largest key, while
107           //setting its parent's right child index to the largest key node's
108           //left child index. recursion downward through this function updates
109           //the heights of the nodes it traverses
110           Node& subtree_root = nodes[subtree_root_index];
111           size_t largest_key_node_index = 0;
112           if (subtree_root_index == 0) {
113               throw std::logic_error("Expected to find a valid node, but didn't");
114           } else {
115               if (subtree_root.right_index) {
116                   largest_key_node_index =
                          remove_largest_key_node_index(subtree_root.right_index);
117                   subtree_root.num_children--;
118                   subtree_root.update_height(nodes);
119               } else {
120                   largest_key_node_index = subtree_root_index;
121                   subtree_root_index = subtree_root.left_index;
122               }
123           }
124           return largest_key_node_index;
125       }
126       virtual void remove_node(size_t& subtree_root_index) {
127           Node& subtree_root = nodes[subtree_root_index];
128           size_t index_to_delete = subtree_root_index;
129           if (subtree_root.right_index || subtree_root.left_index) {
130               //subtree has at least one child
131               if (subtree_root.right_index)
132                   //replace the root with the smallest-keyed node in the right
                          subtree
133                   subtree_root_index =
                          remove_smallest_key_node_index(subtree_root.right_index);
134               else if (subtree_root.left_index)
135                   //replace the root with the largest-keyed node in the left subtree
136                   subtree_root_index =
                          remove_largest_key_node_index(subtree_root.left_index);
137               //have the new root adopt the old root's children
138               Node& new_root = nodes[subtree_root_index];
```

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

```
                    subtree_root.update_height(nodes);
                }
                break;
            case 0:
                //found key, remove the node
                found_key = true;
                value = subtree_root.value;
                remove_node(subtree_root_index);
                break;
            default:
                throw std::domain_error("Unexpected compare() function return
                    value");
            }
        }
        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 << "]";
        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--] += " ";
            write_subtree_buffer(subtree_root.right_index,
                                 buffer_lines,
                                 right_child_line_index,
                                 lbound_line_index,
                                 root_line_index);
        }
        //print the left descendents
```

```
235             if (subtree_root.left_index > 0) {
236                 //at least 1 left child
237                 size_t bottom_dashes = 1;
238                 Node const& left_child = nodes[subtree_root.left_index];
239                 if (left_child.right_index > 0) {
240                     //left child has at least 1 right child
241                     Node const& left_right_child = nodes[left_child.right_index];
242                     bottom_dashes += 2 * (1 + left_right_child.num_children);
243                 }
244                 size_t bottom_line_index = root_line_index + 1;
245                 while (bottom_line_index <= root_line_index + bottom_dashes)
246                     buffer_lines[bottom_line_index++] += "| ";
247                 size_t left_child_line_index = bottom_line_index;
248                 buffer_lines[bottom_line_index++] += "+--";
249                 while (bottom_line_index < ubound_line_index)
250                     buffer_lines[bottom_line_index++] += " ";
251                 write_subtree_buffer(subtree_root.left_index,
252                                      buffer_lines,
253                                      left_child_line_index,
254                                      root_line_index + 1,
255                                      ubound_line_index);
256             }
257         }
258         void add_node_to_free_tree(size_t node_index) {
259             nodes[node_index].disable_and_adopt_free_tree(free_index);
260             free_index = node_index;
261         }
262         size_t procure_node(key_type const& key, value_type const& value) {
263             //updates the free index to the first free node's left child (while
                    transforming that first free
264             //node to an enabled node with the specified key/value) and returns the
                    index of what was the last
265             //free index
266             size_t node_index = free_index;
267             free_index = nodes[free_index].left_index;
268             Node& n = nodes[node_index];
269             n.reset_and_enable(key, value);
270             return node_index;
271         }
272         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)
273                                    size_t& subtree_root_index,
274                                    key_type const& key,
275                                    value_type const& value,
276                                    bool& found_key)
277         {
278             if (subtree_root_index == 0) {
279                 //key not found
280                 subtree_root_index = procure_node(key, value);
281             } else {
282                 //parent was not a leaf
283                 //keep going down to the base of the tree
```

```
284             Node& subtree_root = nodes[subtree_root_index];
285             ++nodes_visited;
286             switch (compare(key, subtree_root.key)) {
287             case -1:
288                 //key is less than subtree root's key
289                 nodes_visited = insert_at_leaf(nodes_visited,
                        subtree_root.left_index, key, value, found_key);
290                 if ( ! found_key) {
291                     //given key is unique to the tree, so a new node was added
292                     subtree_root.num_children++;
293                     subtree_root.update_height(nodes);
294                 }
295                 break;
296             case 1:
297                 //key is greater than subtree root's key
298                 nodes_visited = insert_at_leaf(nodes_visited,
                        subtree_root.right_index, key, value, found_key);
299                 if ( ! found_key) {
300                     //given key is unique to the tree, so a new node was added
301                     subtree_root.num_children++;
302                     subtree_root.update_height(nodes);
303                 }
304                 break;
305             case 0:
306                 //found key, replace the value
307                 subtree_root.value = value;
308                 found_key = true;
309                 break;
310             default:
311                 throw std::domain_error("Unexpected compare() function return
                        value");
312             }
313         }
314         return nodes_visited;
315     }
316     void rotate_left(size_t& subtree_root_index) {
317         Node& subtree_root = nodes[subtree_root_index];
318         size_t right_child_index = subtree_root.right_index;
319         Node& right_child = nodes[right_child_index];
320
321         //original root adopts the right child's left subtree
322         subtree_root.right_index = right_child.left_index;
323         //original root adopted a subtree (whose height did not change), so
                update its height
324         subtree_root.update_height(nodes);
325
326         //right child adopts original root and its children
327         right_child.left_index = subtree_root_index;
328         //right child (new root) adopted the original root (whose height has
                been updated), so update its height
329         right_child.update_height(nodes);
```

```
330             //since right child took the subtree root's place, it has the same
                    number of children as the original root
331             right_child.num_children = subtree_root.num_children;
332
333             //root has new children, so update that counter (done after changing the
                    right child's children counter
334             //because that depends on the original root's counter)
335             subtree_root.num_children = 0;
336             if (subtree_root.left_index != 0)
337                 subtree_root.num_children += 1 +
                        nodes[subtree_root.left_index].num_children;
338             if (subtree_root.right_index != 0)
339                 subtree_root.num_children += 1 +
                        nodes[subtree_root.right_index].num_children;
340
341             //set the right child as the new root
342             subtree_root_index = right_child_index;
343         }
344         void rotate_right(size_t& subtree_root_index) {
345             Node& subtree_root = nodes[subtree_root_index];
346             size_t left_child_index = subtree_root.left_index;
347             Node& left_child = nodes[left_child_index];
348
349             //original root adopts the left child's right subtree
350             subtree_root.left_index = left_child.right_index;
351             //original root adopted a subtree (whose height did not change), so
                    update its height
352             subtree_root.update_height(nodes);
353
354             //left child adopts original root and its children
355             left_child.right_index = subtree_root_index;
356             //left child (new root) adopted the original root (whose height has been
                    updated), so update its height
357             left_child.update_height(nodes);
358             //since left child took the subtree root's place, it has the same number
                    of children as the original root
359             left_child.num_children = subtree_root.num_children;
360
361             //root has new children, so update that counter (done after changing the
                    left child's children counter
362             //because that depends on the original root's counter)
363             subtree_root.num_children = 0;
364             if (subtree_root.left_index != 0)
365                 subtree_root.num_children += 1 +
                        nodes[subtree_root.left_index].num_children;
366             if (subtree_root.right_index != 0)
367                 subtree_root.num_children += 1 +
                        nodes[subtree_root.right_index].num_children;
368
369             //set the left child as the new root
370             subtree_root_index = left_child_index;
371         }
```

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

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

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

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

```
560             returns a list indicating the number of leaf nodes at each height (since
                    the RBST doesn't exhibit
561             true clustering, but can have degenerate branches).
562         */
563         virtual priority_queue<hash_utils::ClusterInventory> cluster_distribution()
                {
564             //use an array to count cluster instances, then feed those to a priority
                    queue and return it.
565             priority_queue<ClusterInventory> cluster_pq;
566             if (is_empty()) return cluster_pq;
567             size_t max_height = nodes[root_index].height;
568             size_t cluster_counter[max_height + 1];
569             for (size_t i = 0; i <= max_height; ++i)
570                 cluster_counter[i] = 0;
571             prepare_cluster_distribution(root_index, 1, cluster_counter);
572             for (size_t i = 1; i <= max_height; ++i)
573                 if (cluster_counter[i] > 0) {
574                     ClusterInventory cluster{i, cluster_counter[i]};
575                     cluster_pq.add_to_queue(cluster);
576                 }
577             return cluster_pq;
578         }
579 
580         /*
581             generate a random number, R, (1,size), and starting with the root (node
                    1), do an in-order
582             traversal to find the R-th occupied node; remove that node (adjusting
                    its children accordingly),
583             and return its key.
584         */
585         virtual key_type remove_random() {
586             if (size() == 0) throw std::logic_error("Cant remove from an empty map");
587             size_t ith_node_to_delete = 1 + hash_utils::rand_i(size());
588             key_type key;
589             remove_ith_node_inorder(root_index, ith_node_to_delete, key);
590             return key;
591         }
592     };
593 }
594 
595 #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,
             typename compare_functor = hash_utils::functors::compare_functor>
    class AVL: public BST<key_type, value_type, compare_functor> {
    /*
        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, compare_functor>;
        using typename super::Node;
        int insert_at_leaf(size_t nodes_visited,
                           size_t& subtree_root_index,
                           key_type const& key,
                           value_type 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) {
            size_t largest_key_node_index =
                super::remove_largest_key_node_index(subtree_root_index);
```

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

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

```
            int nodes_visited = do_remove(0, this->root_index, key, value,
                found_key);
            validate_integrity();
            return nodes_visited;
        }
    };
}

#endif
```