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

```cpp
#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)));
                }
            };
```

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

```cpp
                struct quadratic_probe {
                    bool changes_with_probe_attempt() const {
                        return true;
                    }
                    size_t operator()(const char* key, size_t probe_attempt) const {
                        return probe_attempt;
                    }
                };
                struct hash_double {
                private:
                    size_t hash_numeric(size_t numeric) const {
                        size_t hash = numeric % 97; //simple modulus using a prime number (from algorithms in c++)
                        //the second hash may not be zero (will cause an infinite loop).
                        //also, hash must be relatively prime to map_capacity so that every slot can be hit.
                        //since map capacity is a power of two if we use the capacity planner functor,
                        //both properties are attainable by adding one to the hash if it is even (despite what my
                        //7th grade algebra teacher attempted to teach me, I stubbournly consider zero to be an even
                        //integer despite no formal training in number theory)
                        bool is_even = (hash & 1) == 0;
                        if (is_even)
                            ++hash;
                        return hash;
                    }
                public:
                    bool changes_with_probe_attempt() const {
                        return false;
                    }
                    size_t operator()(const char* key, size_t unused) const {
                        size_t numeric = str_to_numeric(key);
                        return hash_numeric(numeric);
                    }
                    size_t operator()(double key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(int key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(std::string key, size_t unused) const {
                        const char* c_key = key.c_str();
                        return operator()(c_key, unused);
                    }
                };
            }
        }
    }
}
```

```

std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory const& rhs) {
    out << "Cluster{size=" << rhs.cluster_size << ", instances=" << rhs.num_instances << "}";
    return out;
}

#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;
            }
        }
    public:
        //take a linked list of cluster descriptors and add each to the priority queue
        priority_queue(size_t preallocation_size = 100): tree(preallocation_size + 1) {
            T empty_item;
            tree.push_back(empty_item);
        }
        priority_queue(priority_queue const& src) {
            tree = src.tree;
            num_items = src.num_items;
        }
        T get_next_item() {
            std::swap(tree[1], tree[num_items]);
            T ret = tree[num_items--];
            fix_down();
            return ret;
        }
        void add_to_queue(T const& item) {
            tree.push_back(item);
            num_items++;
            fix_up(num_items);
        }
        size_t size() {
            return num_items;
        }
        bool empty() {
            return num_items == 0;
        }
    };
}

#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
```

```
                    //found unoccupied slot
                    break;
            }
            return probes_required;
        }
        //all backing array manipulations should go through the following two methods
        void insert_at_index(key_type const& key, value_type const& value, size_t index) {
            Slot& s = slots[index];
            s.key = key;
            s.value = value;
            if ( ! s.is_occupied) {
                s.is_occupied = true;
                ++num_occupied_slots;
            }
        }
        value_type remove_at_index(size_t index) {
            Slot& s = slots[index];
            if (s.is_occupied) {
                s.is_occupied = false;
                --num_occupied_slots;
            }
            return s.value;
        }
    public:
        HashMapOpenAddressing(size_t const min_capacity)
        {
            if (min_capacity == 0) {
                throw std::domain_error("min_capacity must be at least 1");
            }
            cop3530::hash_utils::functors::map_capacity_planner capacity_planner;
            curr_capacity = capacity_planner(min_capacity); //make capacity a power of 2, greater than the minimum capacity
            slots = new Slot[curr_capacity];
        }
        ~HashMapOpenAddressing() {
            delete slots;
        }
        /*
            if there is space available, adds the specified key/value-pair to the hash map and returns true; otherwise
            returns false. If an item already exists in the map with the same key, replace its value.
        */
        bool insert(key_type const& key, value_type const& value) {
            size_t M = capacity();
            if (M == size())
                return false;
            size_t probes_required = search_internal(key);
            size_t index = (hash(key) + probe(probes_required)) % M;
            insert_at_index(key, value, index);
            return true;
        }
        /*
            if there is an item matching key, removes the key/value-pair from the map, stores it's value in value,
            and returns true; otherwise returns false.
        */
        bool remove(key_type const& key, value_type& value) {
            size_t M = capacity();
            size_t probes_required = search_internal(key);
            size_t index = (hash(key) + probe(probes_required)) % M;
            if (slots[index].key != key)
                //key not found
                return false;
            value = remove_at_index(index);
            size_t start_index = index;
            //remove and reinsert items until find unoccupied slot
            for (int i = 1; ; ++i) {
                index = (start_index + probe(i)) % M;
                Slot const& s = slots[index];
                if (s.is_occupied) {
                    remove_at_index(index);
                    insert(s.key, s.value);
                } else {
                    break;
                }
            }
            return true;
        }
        /*
            if there is an item matching key, stores it's value in value,
            and returns true (the item remains in the map); otherwise returns false.
        */
        bool search(key_type const& key, value_type& value) {
            size_t M = capacity();
            size_t probes_required = search_internal(key);
            size_t index = (hash(key) + probe(probes_required)) % M;
            if (slots[index].key != key)
                //key not found
                return false;
            value = slots[index].value;
            return true;
        }
        /*
            removes all items from the map.
        */
        void clear() {
            size_t cap = capacity();
            for (size_t i = 0; i != cap; ++i)
                slots[i].is_occupied = false;
            num_occupied_slots = 0;
        }
        /*
            returns true IFF the map contains no elements.
        */
        bool is_empty() {
            return size() == 0;
        }
        /*
            returns the number of slots in the map.
        */
        size_t capacity() {
            return curr_capacity;
        }
        /*
            returns the number of items actually stored in the map.
        */
        size_t size() {
            return num_occupied_slots;
        }
        /*
            returns the map's load factor (size = load * capacity).
        */
        double load() {
            return static_cast<double>(size()) / capacity();
        }
        /*
            inserts into the ostream, the backing array's contents in sequential order.
            Empty slots shall be denoted by a hyphen, non-empty slots by that item's
            key. [This function will be used for debugging/monitoring].
        */
        std::ostream& print(std::ostream& out) {
            size_t cap = capacity();
            out << '[';
            for (size_t i = 0; i != cap; ++i) {
                if (slots[i].is_occupied) {
                    out << slots[i].key;
                } else {
                    out << "-";
                }
                if (i + 1 < cap)
                    out << '|';
            }
            out << ']';
            return out;
        }

    };
}

#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

```cpp
#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)));
                }
            };
            struct compare_functor {
                int operator()(const char* a, const char* b) const {
                    int cmp = strcmp(a, b);
                    return (cmp < 0 ? -1 :
                                    (cmp > 0 ? 1 : 0));
                }
                int operator()(double a, double b) const {
                    return (a < b ? -1 :
                                   (a > b ? 1 : 0));
                }
                int operator()(std::string const& a, std::string const& b) const {
                    return (a < b ? -1 :
                                   (a > b ? 1 : 0));
                }
                int operator()(int a, int b) const {
                    return (a < b ? -1 :
                                   (a > b ? 1 : 0));
                }
            };
            namespace primary_hashes {
                struct hash_basic {
                //this is such a stupid hash method, but unlike my pathetic attempts at implementing
                //various other hashing methods, it works and is generalizable to all the required key
                //types. together with double hashing it should make for a passable hashing routine.
                public:
                    size_t operator()(const char* key) const {
                        return str_to_numeric(key);
                    }
                    size_t operator()(double key) const {
                        return static_cast<size_t>(std::fmod(key, max_size_t));
                    }
                    size_t operator()(int key) const {
                        return static_cast<size_t>(key);
                    }
                    size_t operator()(std::string const& key) const {
                        const char* c_key = key.c_str();
                        return operator()(c_key);
                    }
                };
            }
            namespace secondary_hashes {
                struct linear_probe {
                    bool changes_with_probe_attempt() const {
                        return false;
                    }
                    size_t operator()(const char* key, size_t probe_attempt) const {
                        return 1;
                    }
                };
                struct quadratic_probe {
                    bool changes_with_probe_attempt() const {
                        return true;
                    }
                    size_t operator()(const char* key, size_t probe_attempt) const {
                        return probe_attempt;
                    }
                };
                struct hash_double {
                private:
                    size_t hash_numeric(size_t numeric) const {
                        size_t hash = numeric % 97; //simple modulus using a prime number (from algorithms in c++)
                        //the second hash may not be zero (will cause an infinite loop).
                        //also, hash must be relatively prime to map_capacity so that every slot can be hit.
                        //since map capacity is a power of two if we use the capacity planner functor,
                        //both properties are attainable by adding one to the hash if it is even (despite what my
                        //7th grade algebra teacher attempted to teach me, I stubbournly consider zero to be an even
                        //integer despite no formal training in number theory)
                        bool is_even = (hash & 1) == 0;
                        if (is_even)
                            ++hash;
                        return hash;
                    }
                public:
                    bool changes_with_probe_attempt() const {
                        return false;
                    }
                    size_t operator()(const char* key, size_t unused) const {
                        size_t numeric = str_to_numeric(key);
                        return hash_numeric(numeric);
                    }
                    size_t operator()(double key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(int key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(std::string key, size_t unused) const {
                        const char* c_key = key.c_str();
                        return operator()(c_key, unused);
                    }
                };
            }
        }
    }
}
```

```

std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory const& rhs) {
    out << "Cluster{size=" << rhs.cluster_size << ", instances=" << rhs.num_instances << "}";
    return out;
}

#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;
            }
        }
    public:
        //take a linked list of cluster descriptors and add each to the priority queue
        priority_queue(size_t preallocation_size = 100): tree(preallocation_size + 1) {
            T empty_item;
            tree.push_back(empty_item);
        }
        priority_queue(priority_queue const& src) {
            tree = src.tree;
            num_items = src.num_items;
        }
        T get_next_item() {
            std::swap(tree[1], tree[num_items]);
            T ret = tree[num_items--];
            fix_down();
            return ret;
        }
        void add_to_queue(T const& item) {
            tree.push_back(item);
            num_items++;
            fix_up(num_items);
        }
        size_t size() {
            return num_items;
        }
        bool empty() {
            return num_items == 0;
        }
    };
}

#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
            key. if found, stores a reference to that item and returns P, the number of
            probe attempts needed to get to the item (ie the number of chain links needed
            to be traversed). otherwise return -1 * P and stores the pointer to the tail dummy node in
            item_ptr.
        */
        int search_internal(key_type const& key, link& item_ptr) {
            int probe_attempts = 1;
            size_t hash_val = hash(key);
            Bucket& bucket = buckets[hash_val];
            item_ptr = bucket.head;
            while ( ! item_ptr->is_dummy) {
                if (item_ptr->key == key) {
                    //found the key
                    return probe_attempts;
                }
                item_ptr = item_ptr->next;
                ++probe_attempts;
            }
            //key not found
            return probe_attempts * -1;
        }
        void init() {
            buckets = new Bucket[num_buckets];
            num_items = 0;
        }
    public:
        HashMapBuckets(size_t const min_buckets)
        {
            if (min_buckets == 0) {
                throw std::domain_error("min_buckets must be at least 1");
            }
            cop3530::hash_utils::functors::map_capacity_planner capacity_planner;
            num_buckets = capacity_planner(min_buckets); //make capacity a power of 2, greater than the minimum capacity
            init();
        }
        ~HashMapBuckets() {
            delete[] buckets;
        }
        /*
            if there is space available, adds the specified key/value-pair to the hash map and returns true; otherwise
            returns false. If an item already exists in the map with the same key, replace its value.
            note: this will never return false because we add to a linked list to resolve collisions
        */
        bool insert(key_type const& key, value_type const& value) {
            Item* item;
            int probes_required = search_internal(key, item);
            if (probes_required > 0)
                //found item
                item->value = value;
            else {
                //currently holding tail (item not found). transform it into a valid item then add a new tail
                item->is_dummy = false;
                item->key = key;
                item->value = value;
                item->next = new Item(nullptr);
                ++num_items;
            }
            return true;
        }
        /*
            if there is an item matching key, removes the key/value-pair from the map, stores it's
            value in value, and returns true; otherwise returns false.
        */
        bool remove(key_type const& key, value_type& value) {
            Item* item;
            int probes_required = search_internal(key, item);
            if (probes_required > 0) {
                //found item
                value = item->value;
                //swap the current item for the next one
                Item* to_delete = item->next;
                *item = *to_delete;
                delete to_delete;
                --num_items;
                return true;
            }
            return false;
        }
        /*
            if there is an item matching key, stores it's value in value, and returns true (the
            item remains in the map); otherwise returns false.
        */
        bool search(key_type const& key, value_type& value) {
            Item* item;
            int probes_required = search_internal(key, item);
            if (probes_required > 0) {
                //found item
                value = item->value;
                return true;
            }
            return false;
        }
        /*
            removes all items from the map.
```

```cpp
        */
        void clear() {
            delete buckets;
            init();
        }
        /*
            returns true IFF the map contains no elements.
        */
        bool is_empty() {
            return size() == 0;
        }
        /*
            returns the number of slots in the map.
        */
        size_t capacity() {
            return num_buckets;
        }
        /*
            returns the number of items actually stored in the map.
        */
        size_t size() {
            return num_items;
        }
        /*
            returns the map's load factor (occupied buckets = load * capacity).
        */
        double load() {
            size_t occupied_buckets = 0;
            if (size() > 0) {
                size_t M = capacity();
                for (size_t i = 0; i != M; ++i) {
                    Bucket const& bucket = buckets[i];
                    if ( ! bucket.head->is_dummy)
                        //bucket has at least one item
                        occupied_buckets++;
                }
            }
            return static_cast<double>(occupied_buckets) / capacity();
        }
        /*
            inserts into the ostream, the backing array's contents in sequential order.
            Empty slots shall be denoted by a hyphen, non-empty slots by that item's
            key. [This function will be used for debugging/monitoring].
        */
        std::ostream& print(std::ostream& out) {
            size_t cap = capacity();
            bool print_separator = false;
            out << '[';
            for (size_t i = 0; i != cap; ++i) {
                Bucket const& bucket = buckets[i];
                for (Item* item = bucket.head; item->is_dummy != true; item = item->next) {
                    if (print_separator)
                        out << "|";
                    else
                        print_separator = true;
                    out << item->key;
                }
            }
            out << ’]’;
            return out;
        }

    };
}

#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)));
                }
            };
```

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

```
                struct quadratic_probe {
                    bool changes_with_probe_attempt() const {
                        return true;
                    }
                    size_t operator()(const char* key, size_t probe_attempt) const {
                        return probe_attempt;
                    }
                };
                struct hash_double {
                private:
                    size_t hash_numeric(size_t numeric) const {
                        size_t hash = numeric % 97; //simple modulus using a prime number (from algorithms in c++)
                        //the second hash may not be zero (will cause an infinite loop).
                        //also, hash must be relatively prime to map_capacity so that every slot can be hit.
                        //since map capacity is a power of two if we use the capacity planner functor,
                        //both properties are attainable by adding one to the hash if it is even (despite what my
                        //7th grade algebra teacher attempted to teach me, I stubbournly consider zero to be an even
                        //integer despite no formal training in number theory)
                        bool is_even = (hash & 1) == 0;
                        if (is_even)
                            ++hash;
                        return hash;
                    }
                public:
                    bool changes_with_probe_attempt() const {
                        return false;
                    }
                    size_t operator()(const char* key, size_t unused) const {
                        size_t numeric = str_to_numeric(key);
                        return hash_numeric(numeric);
                    }
                    size_t operator()(double key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(int key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(std::string key, size_t unused) const {
                        const char* c_key = key.c_str();
                        return operator()(c_key, unused);
                    }
                };
            }
        }
    }
}
```

```

std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory const& rhs) {
    out << "Cluster{size=" << rhs.cluster_size << ", instances=" << rhs.num_instances << "}";
    return out;
}

#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;
            }
        }
    public:
        //take a linked list of cluster descriptors and add each to the priority queue
        priority_queue(size_t preallocation_size = 100): tree(preallocation_size + 1) {
            T empty_item;
            tree.push_back(empty_item);
        }
        priority_queue(priority_queue const& src) {
            tree = src.tree;
            num_items = src.num_items;
        }
        T get_next_item() {
            std::swap(tree[1], tree[num_items]);
            T ret = tree[num_items--];
            fix_down();
            return ret;
        }
        void add_to_queue(T const& item) {
            tree.push_back(item);
            num_items++;
            fix_up(num_items);
        }
        size_t size() {
            return num_items;
        }
        bool empty() {
            return num_items == 0;
        }
    };
}

#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;
        /*
```

```
            searches the map for an item matching key. returns the number of probe attempts needed
            to reach either the item or an empty slot
        */
        int search_internal(Key const& key) {
            size_t M = capacity();
            size_t probes_required;
            for (probes_required = 0; probes_required != M; ++probes_required) {
                size_t slot_index = key.hash(M, probes_required);
                if (slots[slot_index].is_occupied) {
                    if (slots[slot_index].item.key == key) {
                        //found the key
                        break;
                    }
                } else
                    //found unoccupied slot
                    break;
            }
            return probes_required;
        }

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

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

        /*
            if there is an item matching key, removes the key/value-pair from the map, stores it's value in
            value, and returns the number of probes required, P; otherwise returns -1 * P.
        */
        int remove(key_type const& key, value_type& value) {
            size_t M = capacity();
            Key k(key);
            size_t probes_required = search_internal(k);
            size_t index = k.hash(M, probes_required);
            if (slots[index].is_occupied == false || slots[index].item.key != key)
                //key not found
                return -1 * probes_required;
            Value v = remove_at_index(index);
            value = v.raw();
            //remove and reinsert items until find unoccupied slot (guaranteed to happen since we just removed an item)
            for (int i = 1; ; ++i) {
                index = k.hash(M, i);
                Slot const& s = slots[index];
                if (s.is_occupied) {
                    remove_at_index(index);
                    insert(s.item.key.raw(), s.item.value.raw());
                } else {
                    break;
                }
            }
            return probes_required;
        }

        /*
```

```
            if there is an item matching key, stores it's value in value, and returns the
            number of probes required, P; otherwise returns -1 * P. Regardless, the item
            remains in the map.
        */
        int search(key_type const& key, value_type& value) {
            size_t M = capacity();
            Key k(key);
            size_t probes_required = search_internal(k);
            size_t index = k.hash(M, probes_required);
            if (slots[index].is_occupied == false || slots[index].item.key != key)
                //key not found
                return -1 * probes_required;
            value = slots[index].item.value.raw();
            return probes_required;
        }

        /*
            removes all items from the map.
        */
        void clear() {
            size_t cap = capacity();
            for (size_t i = 0; i != cap; ++i)
                slots[i].is_occupied = false;
            num_occupied_slots = 0;
        }
        /*
            returns true IFF the map contains no elements.
        */
        bool is_empty() const {
            return size() == 0;
        }
        /*
            returns the number of slots in the map.
        */
        size_t capacity() const {
            return curr_capacity;
        }
        /*
            returns the number of items actually stored in the map.
        */
        size_t size() const {
            return num_occupied_slots;
        }
        /*
            returns the map's load factor (size = load * capacity).
        */
        double load() const {
            return static_cast<double>(size()) / capacity();
        }
        /*
            inserts into the ostream, the backing array's contents in sequential order.
            Empty slots shall be denoted by a hyphen, non-empty slots by that item's
            key. [This function will be used for debugging/monitoring].
        */
        std::ostream& print(std::ostream& out) const {
            size_t cap = capacity();
            out << '[';
            for (size_t i = 0; i != cap; ++i) {
                if (slots[i].is_occupied) {
                    out << slots[i].item.key.raw();
                } else {
                    out << "-";
                }
                if (i + 1 < cap)
                    out << '|';
            }
            out << ']';
            return out;
        }

        priority_queue<ClusterInventory> cluster_distribution() {
            //use an array to count cluster instances, then feed those to a priority queue and return it.
            priority_queue<ClusterInventory> cluster_pq;
            if (size() == 0) return cluster_pq;
            size_t M = capacity();
            size_t cluster_counter[M + 1];
            for (size_t i = 0; i <= M; ++i)
                cluster_counter[i] = 0;
            if (size() == M) {
                //handle the special case when the map is full
                cluster_counter[size()]++;
            } else {
                //have at least one unoccupied slot
                bool first_cluster_skipped = false;
                size_t curr_cluster_size = 0;
                //treat the backing array as a circular buffer and make a maximum of two passes to
                //capture everything, including the wraparound cluster if it exists
                for (size_t i = 1; i != M * 2; ++i) {
                    Slot const& curr_slot = slots[i % M], prev_slot = slots[(i - 1) % M];
                    if (curr_slot.is_occupied && prev_slot.is_occupied) {
                        //still in a cluster
                        ++curr_cluster_size;
                    } else if (curr_slot.is_occupied && prev_slot.is_occupied == false) {
                        //found a new cluster
                        curr_cluster_size = 1;
                    } else if ( ! curr_slot.is_occupied && prev_slot.is_occupied) {
                        //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)));
                }
            };
        struct compare_functor {
            int operator()(const char* a, const char* b) const {
                int cmp = strcmp(a, b);
                return (cmp < 0 ? -1 :
                                  (cmp > 0 ? 1 : 0));
            }
            int operator()(double a, double b) const {
                return (a < b ? -1 :
                                (a > b ? 1 : 0));
            }
            int operator()(std::string const& a, std::string const& b) const {
                return (a < b ? -1 :
                                (a > b ? 1 : 0));
            }
            int operator()(int a, int b) const {
                return (a < b ? -1 :
                                (a > b ? 1 : 0));
            }
        };
        namespace primary_hashes {
            struct hash_basic {
            //this is such a stupid hash method, but unlike my pathetic attempts at implementing
            //various other hashing methods, it works and is generalizable to all the required key
            //types. together with double hashing it should make for a passable hashing routine.
            public:
                size_t operator()(const char* key) const {
                    return str_to_numeric(key);
                }
                size_t operator()(double key) const {
                    return static_cast<size_t>(std::fmod(key, max_size_t));
                }
                size_t operator()(int key) const {
                    return static_cast<size_t>(key);
                }
                size_t operator()(std::string const& key) const {
                    const char* c_key = key.c_str();
                    return operator()(c_key);
                }
            };
        }
        namespace secondary_hashes {
            struct linear_probe {
                bool changes_with_probe_attempt() const {
                    return false;
                }
                size_t operator()(const char* key, size_t probe_attempt) const {
                    return 1;
                }
            };
```

```
                struct quadratic_probe {
                    bool changes_with_probe_attempt() const {
                        return true;
                    }
                    size_t operator()(const char* key, size_t probe_attempt) const {
                        return probe_attempt;
                    }
                };
                struct hash_double {
                private:
                    size_t hash_numeric(size_t numeric) const {
                        size_t hash = numeric % 97; //simple modulus using a prime number (from algorithms in c++)
                        //the second hash may not be zero (will cause an infinite loop).
                        //also, hash must be relatively prime to map_capacity so that every slot can be hit.
                        //since map capacity is a power of two if we use the capacity planner functor,
                        //both properties are attainable by adding one to the hash if it is even (despite what my
                        //7th grade algebra teacher attempted to teach me, I stubbournly consider zero to be an even
                        //integer despite no formal training in number theory)
                        bool is_even = (hash & 1) == 0;
                        if (is_even)
                            ++hash;
                        return hash;
                    }
                public:
                    bool changes_with_probe_attempt() const {
                        return false;
                    }
                    size_t operator()(const char* key, size_t unused) const {
                        size_t numeric = str_to_numeric(key);
                        return hash_numeric(numeric);
                    }
                    size_t operator()(double key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(int key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(std::string key, size_t unused) const {
                        const char* c_key = key.c_str();
                        return operator()(c_key, unused);
                    }
                };
            }
        }
    }
}
```

```cpp

std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory const& rhs) {
    out << "Cluster{size=" << rhs.cluster_size << ", instances=" << rhs.num_instances << "}";
    return out;
}

#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;
            }
        }
    public:
        //take a linked list of cluster descriptors and add each to the priority queue
        priority_queue(size_t preallocation_size = 100): tree(preallocation_size + 1) {
            T empty_item;
            tree.push_back(empty_item);
        }
        priority_queue(priority_queue const& src) {
            tree = src.tree;
            num_items = src.num_items;
        }
        T get_next_item() {
            std::swap(tree[1], tree[num_items]);
            T ret = tree[num_items--];
            fix_down();
            return ret;
        }
        void add_to_queue(T const& item) {
            tree.push_back(item);
            num_items++;
            fix_up(num_items);
        }
        size_t size() {
            return num_items;
        }
        bool empty() {
            return num_items == 0;
        }
    };
}

#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;
            }
            ~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;
        /*
            searches the bucket corresponding to the specified key's hash for that
            key. if found, stores a reference to that item and returns P, the number of
            probe attempts needed to get to the item (ie the number of chain links needed
            to be traversed). otherwise return -1 * P and stores the pointer to the tail dummy node in
            item_ptr.
        */
        int search_internal(Key const& key, link& item_ptr) {
            int probe_attempts = 1;
            size_t hash_val = key.hash(capacity(), 0);
            Bucket& bucket = buckets[hash_val];
            item_ptr = bucket.head;
            while ( ! item_ptr->is_dummy) {
                if (item_ptr->key == key) {
                    //found the key
                    return probe_attempts;
                }
                item_ptr = item_ptr->next;
                ++probe_attempts;
            }
            //key not found
            return probe_attempts * -1;
        }
        void init() {
            buckets = new Bucket[num_buckets];
            num_items = 0;
        }
    public:
        HashMapBucketsGeneric(size_t const min_buckets)
        {
            if (min_buckets == 0) {
                throw std::domain_error("min_buckets must be at least 1");
            }
            cop3530::hash_utils::functors::map_capacity_planner capacity_planner;
            num_buckets = capacity_planner(min_buckets); //make capacity a power of 2, greater than the minimum capacity
            init();
        }
        ~HashMapBucketsGeneric() {
            delete[] buckets;
        }
        /*
            if there is space available, adds the specified key/value-pair to the hash map and returns the
            number of probes required, P; otherwise returns -1 * P (that's a lie: we will always have space
            available because each bucket contains a linked list that is indefinitely growable). If an item
            already exists in the map with the same key, replace its value.
        */
        int insert(key_type const& key, value_type const& value) {
            Item* item;
            Key k(key);
            Value v(value);
            int probes_required = search_internal(k, item);
            if (probes_required > 0)
                //found item
                item->value = v;
            else {
                //currently holding tail (item not found). transform it into a valid item then add a new tail
                item->is_dummy = false;
                item->key = k;
                item->value = v;
                item->next = new Item(nullptr);
                ++num_items;
            }
            return std::abs(probes_required);
        }
        /*
            if there is an item matching key, removes the key/value-pair from the map, stores it's value in
            value, and returns the number of probes required, P; otherwise returns -1 * P.
        */
        int remove(key_type const& key, value_type& value) {
            Key k(key);
            Item* item;
            int probes_required = search_internal(key, item);
            if (probes_required > 0) {
                //found item
                value = item->value.raw();
                //swap the current item for the next one
                Item* to_delete = item->next;
                *item = *to_delete;
                delete to_delete;
                --num_items;
            }
            return probes_required;
        }
        /*
            if there is an item matching key, stores it’s value in value, and returns the
            number of probes required, P; otherwise returns -1 * P. Regardless, the item
            remains in the map.
        */
        int search(key_type const& key, value_type& value) {
            Item* item;
            Key k(key);
            int probes_required = search_internal(k, item);
            if (probes_required > 0) {
                //found item
                value = item->value.raw();
            }
            return probes_required;
        }
        /*
            removes all items from the map.
        */
        void clear() {
            delete buckets;
            init();
        }
        /*
            returns true IFF the map contains no elements.
        */
        bool is_empty() {
            return size() == 0;
        }
        /*
            returns the number of slots in the map.
        */
        size_t capacity() {
            return num_buckets;
        }
        /*
            returns the number of items actually stored in the map.
        */
        size_t size() {
            return num_items;
        }
        /*
            returns the map’s load factor (occupied buckets = load * capacity).
        */
        double load() {
            size_t occupied_buckets = 0;
            if (size() > 0) {
                size_t M = capacity();
                for (size_t i = 0; i != M; ++i) {
                    Bucket const& bucket = buckets[i];
                    if ( ! bucket.head->is_dummy)
                        //bucket has at least one item
                        occupied_buckets++;
                }
            }
            return static_cast<double>(occupied_buckets) / capacity();
        }
        /*
            inserts into the ostream, the backing array's contents in sequential order.
            Empty slots shall be denoted by a hyphen, non-empty slots by that item's
            key. [This function will be used for debugging/monitoring].
        */
        std::ostream& print(std::ostream& out) {
            size_t cap = capacity();
            bool print_separator = false;
            out << '[';
            for (size_t i = 0; i != cap; ++i) {
                Bucket const& bucket = buckets[i];
                for (Item* item = bucket.head; item->is_dummy != true; item = item->next) {
                    if (print_separator)
                        out << "|";
                    else
                        print_separator = true;
                    out << item->key.raw();
                }
            }
            out << ']';
            return out;
        }

        /*
            returns a priority queue containing cluster sizes and instances (in the form of ClusterInventory
            struct instances), sorted by cluster size.
        */
        priority_queue<ClusterInventory> cluster_distribution() {
            //use a simple linked list to count cluster instances, then feed those to a priority queue and return it.
            priority_queue<ClusterInventory> cluster_pq;
            if (size() == 0) return cluster_pq;
            SSLL<ClusterInventory> clusters;
            size_t M = capacity();
            for (size_t i = 0; i != M; ++i) {
                Bucket const& bucket = buckets[i];
                size_t bucket_size = 0;
                Item* item_ptr = bucket.head;
                while ( ! item_ptr->is_dummy) {
                    ++bucket_size;
                    item_ptr = item_ptr->next;
                }
                //I don’t love this O(N^2) implementation, but premature optimization is the root of all evil and late projects
                SSLL<ClusterInventory>::iterator cluster_iterator = clusters.begin();
                SSLL<ClusterInventory>::iterator cluster_iterator_end = clusters.end();
                bool found_cluster = false;
                for (; cluster_iterator != cluster_iterator_end; ++cluster_iterator) {
                    if (cluster_iterator->cluster_size == bucket_size) {
                        found_cluster = true;
                        break;
                    }
                }
                if (found_cluster)
                    cluster_iterator->num_instances++;
                else
                    clusters.push_back({bucket_size, 1});
            }
            SSLL<ClusterInventory>::const_iterator cluster_iterator = clusters.begin();
            SSLL<ClusterInventory>::const_iterator cluster_iterator_end = clusters.end();
            for (; cluster_iterator != cluster_iterator_end; ++cluster_iterator) {
                if (cluster_iterator->cluster_size > 0)
                    cluster_pq.add_to_queue(*cluster_iterator);
            }
            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) {
                Bucket const& bucket = buckets[i];
                Item* item_ptr = bucket.head;
                while ( ! item_ptr->is_dummy) {
                    if (--ith_node_to_delete == 0) {
                        key_type key = item_ptr->key.raw();
                        value_type val_dummy;
                        remove(key, val_dummy);
                        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)));
                }
            };
        struct compare_functor {
            int operator()(const char* a, const char* b) const {
                int cmp = strcmp(a, b);
                return (cmp < 0 ? -1 :
                                (cmp > 0 ? 1 : 0));
            }
            int operator()(double a, double b) const {
                return (a < b ? -1 :
                               (a > b ? 1 : 0));
            }
            int operator()(std::string const& a, std::string const& b) const {
                return (a < b ? -1 :
                               (a > b ? 1 : 0));
            }
            int operator()(int a, int b) const {
                return (a < b ? -1 :
                               (a > b ? 1 : 0));
            }
        };
        namespace primary_hashes {
            struct hash_basic {
            //this is such a stupid hash method, but unlike my pathetic attempts at implementing
            //various other hashing methods, it works and is generalizable to all the required key
            //types. together with double hashing it should make for a passable hashing routine.
            public:
                size_t operator()(const char* key) const {
                    return str_to_numeric(key);
                }
                size_t operator()(double key) const {
                    return static_cast<size_t>(std::fmod(key, max_size_t));
                }
                size_t operator()(int key) const {
                    return static_cast<size_t>(key);
                }
                size_t operator()(std::string const& key) const {
                    const char* c_key = key.c_str();
                    return operator()(c_key);
                }
            };
        }
        namespace secondary_hashes {
            struct linear_probe {
                bool changes_with_probe_attempt() const {
                    return false;
                }
                size_t operator()(const char* key, size_t probe_attempt) const {
                    return 1;
                }
            };
```

```
                struct quadratic_probe {
                    bool changes_with_probe_attempt() const {
                        return true;
                    }
                    size_t operator()(const char* key, size_t probe_attempt) const {
                        return probe_attempt;
                    }
                };
                struct hash_double {
                private:
                    size_t hash_numeric(size_t numeric) const {
                        size_t hash = numeric % 97; //simple modulus using a prime number (from algorithms in c++)
                        //the second hash may not be zero (will cause an infinite loop).
                        //also, hash must be relatively prime to map_capacity so that every slot can be hit.
                        //since map capacity is a power of two if we use the capacity planner functor,
                        //both properties are attainable by adding one to the hash if it is even (despite what my
                        //7th grade algebra teacher attempted to teach me, I stubbournly consider zero to be an even
                        //integer despite no formal training in number theory)
                        bool is_even = (hash & 1) == 0;
                        if (is_even)
                            ++hash;
                        return hash;
                    }
                public:
                    bool changes_with_probe_attempt() const {
                        return false;
                    }
                    size_t operator()(const char* key, size_t unused) const {
                        size_t numeric = str_to_numeric(key);
                        return hash_numeric(numeric);
                    }
                    size_t operator()(double key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(int key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(std::string key, size_t unused) const {
                        const char* c_key = key.c_str();
                        return operator()(c_key, unused);
                    }
                };
            }
        }
    }
}
```

```

std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory
     const& rhs) {
    out << "Cluster{size=" << rhs.cluster_size << ", instances=" <<
         rhs.num_instances << "}";
    return out;
}

#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;
            }
        }
    public:
        //take a linked list of cluster descriptors and add each to the priority queue
        priority_queue(size_t preallocation_size = 100): tree(preallocation_size + 1) {
            T empty_item;
            tree.push_back(empty_item);
        }
        priority_queue(priority_queue const& src) {
            tree = src.tree;
            num_items = src.num_items;
        }
        T get_next_item() {
            std::swap(tree[1], tree[num_items]);
            T ret = tree[num_items--];
            fix_down();
            return ret;
        }
        void add_to_queue(T const& item) {
            tree.push_back(item);
            num_items++;
            fix_up(num_items);
        }
        size_t size() {
            return num_items;
        }
        bool empty() {
            return num_items == 0;
        }
    };
}

#endif // _PRIORITY_QUEUE_H_
```

part4/source/bst.h

part4/source/bst.h

```cpp
#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;
                if (right_index)
                    right_height = nodes[right_index].height;
                height = 1 + std::max(left_height, right_height);
                //todo: delete the following expensive check, or move it into DEBUG condition
                size_t calculated_height = get_height_recursive(nodes);
                if (calculated_height != height) {
                    std::ostringstream msg;
                    msg << "Manually calculated height, " << calculated_height << ", different than tracked height, " << height;
                    throw std::runtime_error(msg.str());
                }
            }
            void disable_and_adopt_free_tree(size_t free_index) {
                is_occupied = false;
                height = 0;
                num_children = 0;
                right_index = 0;
                left_index = free_index;
            }
            void reset_and_enable(key_type const new_key, value_type const& new_value) {
                is_occupied = true;
                height = 1; //self
                left_index = right_index = 0;
                num_children = 0;
                key = new_key;
                value = new_value;
            }
            int balance_factor(const Node* nodes) const {
                size_t left_height = 0, right_height = 0;
                if (left_index)
                    left_height = nodes[left_index].height;
                if (right_index)
                    right_height = nodes[right_index].height;
                return static_cast<long int>(left_height) - static_cast<long int>(right_height);
            }
        };
        Node* nodes; //***note: array is 1-based so leaf nodes have child indices set to zero
        size_t free_index;
        size_t root_index;
        size_t curr_capacity;
        virtual size_t remove_smallest_key_node_index(size_t& subtree_root_index) {
            //returns the index of the node with the smallest key, while
            //setting its parent's left child index to the smallest key node's
            //right child index. recursion downward through this function updates
            //the heights of the nodes it traverses
            Node& subtree_root = nodes[subtree_root_index];
            size_t smallest_key_node_index = 0;
            if (subtree_root_index == 0) {
                throw std::logic_error("Expected to find a valid node, but didn’t");
            } else {
                if (subtree_root.left_index) {
                    smallest_key_node_index =
                         remove_smallest_key_node_index(subtree_root.left_index);
                    subtree_root.num_children--;
                    subtree_root.update_height(nodes);
                } else {
                    smallest_key_node_index = subtree_root_index;
                    subtree_root_index = subtree_root.right_index;
                }
            }
            return smallest_key_node_index;
        }
        virtual size_t remove_largest_key_node_index(size_t& subtree_root_index) {
            //returns the index of the node with the largest key, while
            //setting its parent’s right child index to the largest key node’s
            //left child index. recursion downward through this function updates
            //the heights of the nodes it traverses
            Node& subtree_root = nodes[subtree_root_index];
            size_t largest_key_node_index = 0;
            if (subtree_root_index == 0) {
                throw std::logic_error("Expected to find a valid node, but didn’t");
            } else {
                if (subtree_root.right_index) {
                    largest_key_node_index =
                         remove_largest_key_node_index(subtree_root.right_index);
                    subtree_root.num_children--;
                    subtree_root.update_height(nodes);
                } else {
                    largest_key_node_index = subtree_root_index;
                    subtree_root_index = subtree_root.left_index;
                }
            }
            return largest_key_node_index;
        }
        virtual void remove_node(size_t& subtree_root_index) {
            Node& subtree_root = nodes[subtree_root_index];
            size_t index_to_delete = subtree_root_index;
            if (subtree_root.right_index || subtree_root.left_index) {
                //subtree has at least one child
                if (subtree_root.right_index)
                    //replace the root with the smallest-keyed node in the right
                         subtree
                    subtree_root_index =
                         remove_smallest_key_node_index(subtree_root.right_index);
                else if (subtree_root.left_index)
                    //replace the root with the largest-keyed node in the left subtree
                    subtree_root_index =
                         remove_largest_key_node_index(subtree_root.left_index);
                //have the new root adopt the old root’s children
                Node& new_root = nodes[subtree_root_index];
                new_root.left_index = subtree_root.left_index;
                new_root.right_index = subtree_root.right_index;
                //the new root has the same number of children as the old root, minus one
                new_root.num_children = subtree_root.num_children - 1;
                //removing the smallest/largest-keyed node from the old root has the effect of
                //updating the heights of the old root's relevant subtrees (which the new root
                //just adopted), so we can update the new root's height now
                new_root.update_height(nodes);
            } else
                //neither subtree exists, so just delete the node
                subtree_root_index = 0;
            //node has been disowned by all ancestors, and has disowned all descendents, so free it
            add_node_to_free_tree(index_to_delete);
        }
        virtual int do_remove(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& value,
                            bool& found_key)
        {
            if (subtree_root_index == 0)
                //key not found
                nodes_visited *= -1;
            else {
                Node& subtree_root = nodes[subtree_root_index];
                ++nodes_visited;
                //keep going down to the base of the tree
                switch (compare(key, subtree_root.key)) {
                case -1:
                    //key is less than subtree root's key
                    nodes_visited = do_remove(nodes_visited, subtree_root.left_index, key, value, found_key);
                    if (found_key) {
                        //found the desired node and delete it
                        subtree_root.num_children--;
                        //left child changed, so recompute subtree height
                        subtree_root.update_height(nodes);
                    }
                    break;
                case 1:
                    //key is greater than subtree root's key
                    nodes_visited = do_remove(nodes_visited, subtree_root.right_index, key, value, found_key);
                    if (found_key) {
                        //found the desired node and delete it
                        subtree_root.num_children--;
                        //right child changed, so recompute subtree height
                        subtree_root.update_height(nodes);
                    }
                    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
```

```
            if (subtree_root.left_index > 0) {
                //at least 1 left child
                size_t bottom_dashes = 1;
                Node const& left_child = nodes[subtree_root.left_index];
                if (left_child.right_index > 0) {
                    //left child has at least 1 right child
                    Node const& left_right_child = nodes[left_child.right_index];
                    bottom_dashes += 2 * (1 + left_right_child.num_children);
                }
                size_t bottom_line_index = root_line_index + 1;
                while (bottom_line_index <= root_line_index + bottom_dashes)
                    buffer_lines[bottom_line_index++] += "| ";
                size_t left_child_line_index = bottom_line_index;
                buffer_lines[bottom_line_index++] += "+--";
                while (bottom_line_index < ubound_line_index)
                    buffer_lines[bottom_line_index++] += " ";
                write_subtree_buffer(subtree_root.left_index,
                                   buffer_lines,
                                   left_child_line_index,
                                   root_line_index + 1,
                                   ubound_line_index);
            }
        }
        void add_node_to_free_tree(size_t node_index) {
            nodes[node_index].disable_and_adopt_free_tree(free_index);
            free_index = node_index;
        }
        size_t procure_node(key_type const& key, value_type const& value) {
            //updates the free index to the first free node's left child (while transforming that first free
            //node to an enabled node with the specified key/value) and returns the index of what was the last
            //free index
            size_t node_index = free_index;
            free_index = nodes[free_index].left_index;
            Node& n = nodes[node_index];
            n.reset_and_enable(key, value);
            return node_index;
        }
        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)
                                size_t& subtree_root_index,
                                key_type const& key,
                                value_type const& value,
                                bool& found_key)
        {
            if (subtree_root_index == 0) {
                //key not found
                subtree_root_index = procure_node(key, value);
            } else {
                //parent was not a leaf
                //keep going down to the base of the tree
                Node& subtree_root = nodes[subtree_root_index];
                ++nodes_visited;
                switch (compare(key, subtree_root.key)) {
                case -1:
                    //key is less than subtree root's key
                    nodes_visited = insert_at_leaf(nodes_visited,
                         subtree_root.left_index, key, value, found_key);
                    if ( ! found_key) {
                        //given key is unique to the tree, so a new node was added
                        subtree_root.num_children++;
                        subtree_root.update_height(nodes);
                    }
                    break;
                case 1:
                    //key is greater than subtree root's key
                    nodes_visited = insert_at_leaf(nodes_visited,
                         subtree_root.right_index, key, value, found_key);
                    if ( ! found_key) {
                        //given key is unique to the tree, so a new node was added
                        subtree_root.num_children++;
                        subtree_root.update_height(nodes);
                    }
                    break;
                case 0:
                    //found key, replace the value
                    subtree_root.value = value;
                    found_key = true;
                    break;
                default:
                    throw std::domain_error("Unexpected compare() function return
                         value");
                }
            }
            return nodes_visited;
        }
        void rotate_left(size_t& subtree_root_index) {
            Node& subtree_root = nodes[subtree_root_index];
            size_t right_child_index = subtree_root.right_index;
            Node& right_child = nodes[right_child_index];

            //original root adopts the right child's left subtree
            subtree_root.right_index = right_child.left_index;
            //original root adopted a subtree (whose height did not change), so
                 update its height
            subtree_root.update_height(nodes);

            //right child adopts original root and its children
            right_child.left_index = subtree_root_index;
            //right child (new root) adopted the original root (whose height has
                 been updated), so update its height
            right_child.update_height(nodes);
            //since right child took the subtree root’s place, it has the same number of children as the original root
            right_child.num_children = subtree_root.num_children;

            //root has new children, so update that counter (done after changing the right child’s children counter
            //because that depends on the original root’s counter)
            subtree_root.num_children = 0;
            if (subtree_root.left_index != 0)
                subtree_root.num_children += 1 + nodes[subtree_root.left_index].num_children;
            if (subtree_root.right_index != 0)
                subtree_root.num_children += 1 + nodes[subtree_root.right_index].num_children;

            //set the right child as the new root
            subtree_root_index = right_child_index;
        }
        void rotate_right(size_t& subtree_root_index) {
            Node& subtree_root = nodes[subtree_root_index];
            size_t left_child_index = subtree_root.left_index;
            Node& left_child = nodes[left_child_index];

            //original root adopts the left child’s right subtree
            subtree_root.left_index = left_child.right_index;
            //original root adopted a subtree (whose height did not change), so update its height
            subtree_root.update_height(nodes);

            //left child adopts original root and its children
            left_child.right_index = subtree_root_index;
            //left child (new root) adopted the original root (whose height has been updated), so update its height
            left_child.update_height(nodes);
            //since left child took the subtree root’s place, it has the same number of children as the original root
            left_child.num_children = subtree_root.num_children;

            //root has new children, so update that counter (done after changing the left child’s children counter
            //because that depends on the original root’s counter)
            subtree_root.num_children = 0;
            if (subtree_root.left_index != 0)
                subtree_root.num_children += 1 + nodes[subtree_root.left_index].num_children;
            if (subtree_root.right_index != 0)
                subtree_root.num_children += 1 + nodes[subtree_root.right_index].num_children;

            //set the left child as the new root
            subtree_root_index = left_child_index;
        }
        int do_search(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 value) const
        {
            if (subtree_root_index == 0)
                //key not found
                nodes_visited *= -1;
            else {
                Node const& subtree_root = nodes[subtree_root_index];
                ++nodes_visited;
                switch (compare(key, subtree_root.key)) {
                case -1:
                    //key is less than subtree root key
                    nodes_visited = do_search(nodes_visited, subtree_root.left_index, key, value);
                    break;
                case 1:
                    //key is greater than subtree root key
                    nodes_visited = do_search(nodes_visited, subtree_root.right_index, key, value);
                    break;
                case 0:
                    //found key
                    value = subtree_root.value;
                    break;
                default:
                    throw std::domain_error("Unexpected compare() function return value");
                }
            }
            return nodes_visited;
        }
        void prepare_cluster_distribution(size_t subtree_root_index,
                                      size_t curr_height, //includes the height of the current node, ie assumes current node exists
                                      size_t cluster_counter[])
        {
            Node const& subtree_root = nodes[subtree_root_index];
            if ( ! subtree_root.left_index && ! subtree_root.right_index)
                //at a leaf node
                cluster_counter[curr_height]++;
            else {
                if (subtree_root.left_index)
                    prepare_cluster_distribution(subtree_root.left_index, curr_height + 1, cluster_counter);
                if (subtree_root.right_index)
                    prepare_cluster_distribution(subtree_root.right_index, curr_height + 1, cluster_counter);
            }
        }

        void remove_ith_node_inorder(size_t& subtree_root_index,
                                   size_t& ith_node_to_delete,
                                   key_type& key)
        {
            Node& subtree_root = nodes[subtree_root_index];
            if (subtree_root.left_index)
                remove_ith_node_inorder(subtree_root.left_index, ith_node_to_delete,
                    key);
            if (ith_node_to_delete == 0)
                //deleted node in child subtree; nothing more to do
                return;
            if (--ith_node_to_delete == 0) {
                //delete the current node
                value_type dummy_val;
                remove(subtree_root.key, dummy_val);
                key = subtree_root.key;
                return;
            }
            if (subtree_root.right_index)
                remove_ith_node_inorder(subtree_root.right_index,
                    ith_node_to_delete, key);
        }

    public:
        /*
            The constructor will allocate an array of capacity (binary
            tree) nodes. Then make a chain from all the nodes (e.g.,
            make node 2 the left child of node 1, make node 3 the left
            child of node 2, &c. this is the initial free list.
        */
        BST(size_t capacity):
            curr_capacity(capacity)
        {
            if (capacity == 0) {
                throw std::domain_error("capacity must be at least 1");
            }
            nodes = new Node[capacity + 1];
            clear();
        }
        /*
            if there is space available, adds the specified key/value-pair to the
                tree
            and returns the number of nodes visited, V; otherwise returns -1 * V. If
                an
            item already exists in the tree with the same key, replace its value.
        */
        virtual int insert(key_type const& key, value_type const& value) {
            if (size() == capacity())
                //no more space
                return 0;
            bool found_key = false;
            return insert_at_leaf(0, root_index, key, value, found_key);
        }
        /*
            if there is an item matching key, removes the key/value-pair from the tree, stores
            it's value in value, and returns the number of probes required, V; otherwise returns -1 * V.
        */
        virtual int remove(key_type const& key, value_type& value) {
            bool found_key = false;
            return do_remove(0, root_index, key, value, found_key);
        }
        /*
            if there is an item matching key, stores it's value in value, and returns the number
            of nodes visited, V; otherwise returns -1 * V. Regardless, the item remains in the tree.
        */
        virtual int search(key_type const& key, value_type& value) {
            return do_search(0, root_index, key, value);
        }
        /*
            removes all items from the map
        */
        virtual void clear() {
            //Since I use size_t to hold the node indices, I make the node array
            //1-based, with child index of 0 indicating that the current node is a leaf
            for (size_t i = 1; i != capacity(); ++i)
                nodes[i].disable_and_adopt_free_tree(i + 1);
            free_index = 1;
            root_index = 0;
        }
        /*
            returns true IFF the map contains no elements.
        */
        virtual bool is_empty() const {
            return size() == 0;
        }
        /*
            returns the number of slots in the backing array.
        */
        virtual size_t capacity() const {
            return curr_capacity;
        }
        /*
            returns the number of items actually stored in the tree.
        */
        virtual size_t size() const {
            if (root_index == 0) return 0;
            Node const& root = nodes[root_index];
            return 1 + root.num_children;
        }
        /*
            [not a regular BST operation, but specific to this implementation]
            returns the tree's load factor: load = size / capacity.
        */
        virtual double load() const {
            return static_cast<double>(size()) / capacity();
        }
        /*
            prints the tree in the following format:
            +--[tiger]
            |  |
            |  |  +--[panther]
            |  |  |
            |  +--[ocelot]
            |     |
            |     +--[lion]
            |
            [leopard]
            |
            |     +--[house cat]
            |     |
            |  +--[cougar]
            |  |
            +--[cheetah]
               |
               +--[bobcat]
        */
        virtual std::ostream& print(std::ostream& out) const {
            if (root_index == 0)
                return out;
            size_t num_lines = size() * 2 - 1;
            //use CDAL here so we can print really super-huge trees where the write buffer doesn't fit in memory
            CDAL<std::string> buffer_lines(100000);
            for(size_t i = 0; i <= num_lines; ++i)
                buffer_lines.push_back("");
            Node const& root = nodes[root_index];
            size_t root_line_index = 1;
            if (root.right_index) {
                root_line_index += 2 * (1 + nodes[root.right_index].num_children);
            }
            write_subtree_buffer(root_index, buffer_lines, root_line_index, 1, num_lines + 1);
            for (size_t i = 1; i <= num_lines; ++i)
                out << buffer_lines[i] << std::endl;
            return out;
        }

        /*
```

```
            returns a list indicating the number of leaf nodes at each height (since the RBST doesn’t exhibit
            true clustering, but can have degenerate branches).
        */
        virtual priority_queue<hash_utils::ClusterInventory> cluster_distribution() {
            //use an array to count cluster instances, then feed those to a priority queue and return it.
            priority_queue<ClusterInventory> cluster_pq;
            if (is_empty()) return cluster_pq;
            size_t max_height = nodes[root_index].height;
            size_t cluster_counter[max_height + 1];
            for (size_t i = 0; i <= max_height; ++i)
                cluster_counter[i] = 0;
            prepare_cluster_distribution(root_index, 1, cluster_counter);
            for (size_t i = 1; i <= max_height; ++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 the root (node 1), do an in-order
            traversal to find the R-th occupied node; remove that node (adjusting its children accordingly),
            and return its key.
        */
        virtual key_type remove_random() {
            if (size() == 0) throw std::logic_error("Cant remove from an empty map");
            size_t ith_node_to_delete = 1 + hash_utils::rand_i(size());
            key_type key;
            remove_ith_node_inorder(root_index, ith_node_to_delete, key);
            return key;
        }
    };
}

#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,
                         key_type const& key,
                         value_type const& value,
                         bool& found_key)
        {
            if (subtree_root_index == 0) {
                //parent was a leaf, so create a new leaf
                subtree_root_index = this->procure_node(key, value);
            } else {
                //parent was not a leaf
                Node& subtree_root = this->nodes[subtree_root_index];
                ++nodes_visited;
                //keep going down to the base of the tree
                switch (this->compare(key, subtree_root.key)) {
                case -1:
                    //key is less than subtree root's key
                    nodes_visited = insert_at_root(nodes_visited,
                        subtree_root.left_index, key, value, found_key);
                    if ( ! found_key) {
                        //new node currently a new child of subtree root, so increment
                        //the subtree root's number of children before rotating - new
                            node
                        //will adopt the root and its children and will take on the
                            value
                        //of its num_children
                        subtree_root.num_children++;
                        //current subtree root may have had its height changed, so
                            update that before
                        //promoting the new node
                        subtree_root.update_height(this->nodes);
                        this->rotate_right(subtree_root_index);
                    }
                    break;
                case 1:
                    //key is greater than subtree root's key
                    nodes_visited = insert_at_root(nodes_visited,
                        subtree_root.right_index, key, value, found_key);
                    if ( ! found_key) {
                        subtree_root.num_children++;
                        //current subtree root may have had its height changed, so
                            update that before
                        //promoting the new node
                        subtree_root.update_height(this->nodes);
                        this->rotate_left(subtree_root_index);
                    }
                    break;
                case 0:
                    //found key, replace the value
                    subtree_root.value = value;
                    found_key = true;
                    break;
                default:
                    throw std::domain_error("insert_at_root: Unexpected compare() function return value");
                }
            }
            return nodes_visited;
        }
    public:
        RBST(size_t capacity): super(capacity) {}
        /*
            if there is space available, adds the specified key/value-pair to the tree
            and returns the number of nodes visited, V; otherwise returns -1 * V. If an
            item already exists in the tree with the same key, replace its value.
        */
        int insert(key_type const& key, value_type const& value) {
            if (this->size() == this->capacity())
                //no more space
                return 0;
            bool found_key = false;
            return insert_at_leaf(0, this->root_index, key, value, found_key);
        }
    };
}

#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

```cpp
#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)));
                }
            };
```

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

```
                struct quadratic_probe {
                    bool changes_with_probe_attempt() const {
                        return true;
                    }
                    size_t operator()(const char* key, size_t probe_attempt) const {
                        return probe_attempt;
                    }
                };
                struct hash_double {
                private:
                    size_t hash_numeric(size_t numeric) const {
                        size_t hash = numeric % 97; //simple modulus using a prime number (from algorithms in c++)
                        //the second hash may not be zero (will cause an infinite loop).
                        //also, hash must be relatively prime to map_capacity so that every slot can be hit.
                        //since map capacity is a power of two if we use the capacity planner functor,
                        //both properties are attainable by adding one to the hash if it is even (despite what my
                        //7th grade algebra teacher attempted to teach me, I stubbournly consider zero to be an even
                        //integer despite no formal training in number theory)
                        bool is_even = (hash & 1) == 0;
                        if (is_even)
                            ++hash;
                        return hash;
                    }
                public:
                    bool changes_with_probe_attempt() const {
                        return false;
                    }
                    size_t operator()(const char* key, size_t unused) const {
                        size_t numeric = str_to_numeric(key);
                        return hash_numeric(numeric);
                    }
                    size_t operator()(double key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(int key, size_t unused) const {
                        return hash_numeric(key);
                    }
                    size_t operator()(std::string key, size_t unused) const {
                        const char* c_key = key.c_str();
                        return operator()(c_key, unused);
                    }
                };
            }
        }
    }
}
```

```

std::ostream& operator<<(std::ostream& out, cop3530::hash_utils::ClusterInventory const& rhs) {
    out << "Cluster{size=" << rhs.cluster_size << ", instances=" << rhs.num_instances << "}";
    return out;
}

#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;
            }
        }
    public:
        //take a linked list of cluster descriptors and add each to the priority queue
        priority_queue(size_t preallocation_size = 100): tree(preallocation_size + 1) {
            T empty_item;
            tree.push_back(empty_item);
        }
        priority_queue(priority_queue const& src) {
            tree = src.tree;
            num_items = src.num_items;
        }
        T get_next_item() {
            std::swap(tree[1], tree[num_items]);
            T ret = tree[num_items--];
            fix_down();
            return ret;
        }
        void add_to_queue(T const& item) {
            tree.push_back(item);
            num_items++;
            fix_up(num_items);
        }
        size_t size() {
            return num_items;
        }
        bool empty() {
            return num_items == 0;
        }
    };
}

#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;
                if (right_index)
                    right_height = nodes[right_index].height;
                height = 1 + std::max(left_height, right_height);
                //todo: delete the following expensive check, or move it into DEBUG condition
                size_t calculated_height = get_height_recursive(nodes);
                if (calculated_height != height) {
                    std::ostringstream msg;
                    msg << "Manually calculated height, " << calculated_height << ", different than tracked height, " << height;
                    throw std::runtime_error(msg.str());
                }
            }
            void disable_and_adopt_free_tree(size_t free_index) {
                is_occupied = false;
                height = 0;
                num_children = 0;
                right_index = 0;
                left_index = free_index;
            }
            void reset_and_enable(key_type const new_key, value_type const& new_value) {
                is_occupied = true;
                height = 1; //self
                left_index = right_index = 0;
                num_children = 0;
                key = new_key;
                value = new_value;
            }
            int balance_factor(const Node* nodes) const {
                size_t left_height = 0, right_height = 0;
                if (left_index)
                    left_height = nodes[left_index].height;
                if (right_index)
                    right_height = nodes[right_index].height;
                return static_cast<long int>(left_height) - static_cast<long int>(right_height);
            }
        };
        Node* nodes; //***note: array is 1-based so leaf nodes have child indices set to zero
        size_t free_index;
        size_t root_index;
        size_t curr_capacity;
        virtual size_t remove_smallest_key_node_index(size_t& subtree_root_index) {
            //returns the index of the node with the smallest key, while
            //setting its parent's left child index to the smallest key node's
            //right child index. recursion downward through this function updates
            //the heights of the nodes it traverses
            Node& subtree_root = nodes[subtree_root_index];
            size_t smallest_key_node_index = 0;
            if (subtree_root_index == 0) {
```

```
            throw std::logic_error("Expected to find a valid node, but didn’t");
        } else {
            if (subtree_root.left_index) {
                smallest_key_node_index =
                    remove_smallest_key_node_index(subtree_root.left_index);
                subtree_root.num_children--;
                subtree_root.update_height(nodes);
            } else {
                smallest_key_node_index = subtree_root_index;
                subtree_root_index = subtree_root.right_index;
            }
        }
        return smallest_key_node_index;
    }
    virtual size_t remove_largest_key_node_index(size_t& subtree_root_index) {
        //returns the index of the node with the largest key, while
        //setting its parent’s right child index to the largest key node’s
        //left child index. recursion downward through this function updates
        //the heights of the nodes it traverses
        Node& subtree_root = nodes[subtree_root_index];
        size_t largest_key_node_index = 0;
        if (subtree_root_index == 0) {
            throw std::logic_error("Expected to find a valid node, but didn’t");
        } else {
            if (subtree_root.right_index) {
                largest_key_node_index =
                    remove_largest_key_node_index(subtree_root.right_index);
                subtree_root.num_children--;
                subtree_root.update_height(nodes);
            } else {
                largest_key_node_index = subtree_root_index;
                subtree_root_index = subtree_root.left_index;
            }
        }
        return largest_key_node_index;
    }
    virtual void remove_node(size_t& subtree_root_index) {
        Node& subtree_root = nodes[subtree_root_index];
        size_t index_to_delete = subtree_root_index;
        if (subtree_root.right_index || subtree_root.left_index) {
            //subtree has at least one child
            if (subtree_root.right_index)
                //replace the root with the smallest-keyed node in the right
                    subtree
                subtree_root_index =
                    remove_smallest_key_node_index(subtree_root.right_index);
            else if (subtree_root.left_index)
                //replace the root with the largest-keyed node in the left subtree
                subtree_root_index =
                    remove_largest_key_node_index(subtree_root.left_index);
            //have the new root adopt the old root’s children
            Node& new_root = nodes[subtree_root_index];
```

```
                new_root.left_index = subtree_root.left_index;
                new_root.right_index = subtree_root.right_index;
                //the new root has the same number of children as the old root, minus one
                new_root.num_children = subtree_root.num_children - 1;
                //removing the smallest/largest-keyed node from the old root has the effect of
                //updating the heights of the old root's relevant subtrees (which the new root
                //just adopted), so we can update the new root's height now
                new_root.update_height(nodes);
            } else
                //neither subtree exists, so just delete the node
                subtree_root_index = 0;
            //node has been disowned by all ancestors, and has disowned all descendents, so free it
            add_node_to_free_tree(index_to_delete);
        }
        virtual int do_remove(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& value,
                              bool& found_key)
        {
            if (subtree_root_index == 0)
                //key not found
                nodes_visited *= -1;
            else {
                Node& subtree_root = nodes[subtree_root_index];
                ++nodes_visited;
                //keep going down to the base of the tree
                switch (compare(key, subtree_root.key)) {
                case -1:
                    //key is less than subtree root's key
                    nodes_visited = do_remove(nodes_visited, subtree_root.left_index, key, value, found_key);
                    if (found_key) {
                        //found the desired node and delete it
                        subtree_root.num_children--;
                        //left child changed, so recompute subtree height
                        subtree_root.update_height(nodes);
                    }
                    break;
                case 1:
                    //key is greater than subtree root's key
                    nodes_visited = do_remove(nodes_visited, subtree_root.right_index, key, value, found_key);
                    if (found_key) {
                        //found the desired node and delete it
                        subtree_root.num_children--;
                        //right child changed, so recompute subtree height
                        subtree_root.update_height(nodes);
                    }
                    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
```

```
            if (subtree_root.left_index > 0) {
                //at least 1 left child
                size_t bottom_dashes = 1;
                Node const& left_child = nodes[subtree_root.left_index];
                if (left_child.right_index > 0) {
                    //left child has at least 1 right child
                    Node const& left_right_child = nodes[left_child.right_index];
                    bottom_dashes += 2 * (1 + left_right_child.num_children);
                }
                size_t bottom_line_index = root_line_index + 1;
                while (bottom_line_index <= root_line_index + bottom_dashes)
                    buffer_lines[bottom_line_index++] += "| ";
                size_t left_child_line_index = bottom_line_index;
                buffer_lines[bottom_line_index++] += "+--";
                while (bottom_line_index < ubound_line_index)
                    buffer_lines[bottom_line_index++] += " ";
                write_subtree_buffer(subtree_root.left_index,
                                     buffer_lines,
                                     left_child_line_index,
                                     root_line_index + 1,
                                     ubound_line_index);
            }
        }
        void add_node_to_free_tree(size_t node_index) {
            nodes[node_index].disable_and_adopt_free_tree(free_index);
            free_index = node_index;
        }
        size_t procure_node(key_type const& key, value_type const& value) {
            //updates the free index to the first free node's left child (while transforming that first free
            //node to an enabled node with the specified key/value) and returns the index of what was the last
            //free index
            size_t node_index = free_index;
            free_index = nodes[free_index].left_index;
            Node& n = nodes[node_index];
            n.reset_and_enable(key, value);
            return node_index;
        }
        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)
                                   size_t& subtree_root_index,
                                   key_type const& key,
                                   value_type const& value,
                                   bool& found_key)
        {
            if (subtree_root_index == 0) {
                //key not found
                subtree_root_index = procure_node(key, value);
            } else {
                //parent was not a leaf
                //keep going down to the base of the tree
                Node& subtree_root = nodes[subtree_root_index];
                ++nodes_visited;
                switch (compare(key, subtree_root.key)) {
                case -1:
                    //key is less than subtree root's key
                    nodes_visited = insert_at_leaf(nodes_visited,
                        subtree_root.left_index, key, value, found_key);
                    if ( ! found_key) {
                        //given key is unique to the tree, so a new node was added
                        subtree_root.num_children++;
                        subtree_root.update_height(nodes);
                    }
                    break;
                case 1:
                    //key is greater than subtree root's key
                    nodes_visited = insert_at_leaf(nodes_visited,
                        subtree_root.right_index, key, value, found_key);
                    if ( ! found_key) {
                        //given key is unique to the tree, so a new node was added
                        subtree_root.num_children++;
                        subtree_root.update_height(nodes);
                    }
                    break;
                case 0:
                    //found key, replace the value
                    subtree_root.value = value;
                    found_key = true;
                    break;
                default:
                    throw std::domain_error("Unexpected compare() function return
                        value");
                }
            }
            return nodes_visited;
        }
        void rotate_left(size_t& subtree_root_index) {
            Node& subtree_root = nodes[subtree_root_index];
            size_t right_child_index = subtree_root.right_index;
            Node& right_child = nodes[right_child_index];

            //original root adopts the right child's left subtree
            subtree_root.right_index = right_child.left_index;
            //original root adopted a subtree (whose height did not change), so
                update its height
            subtree_root.update_height(nodes);

            //right child adopts original root and its children
            right_child.left_index = subtree_root_index;
            //right child (new root) adopted the original root (whose height has
                been updated), so update its height
            right_child.update_height(nodes);
```

```cpp
            //since right child took the subtree root's place, it has the same number of children as the original root
            right_child.num_children = subtree_root.num_children;

            //root has new children, so update that counter (done after changing the right child's children counter
            //because that depends on the original root's counter)
            subtree_root.num_children = 0;
            if (subtree_root.left_index != 0)
                subtree_root.num_children += 1 + nodes[subtree_root.left_index].num_children;
            if (subtree_root.right_index != 0)
                subtree_root.num_children += 1 + nodes[subtree_root.right_index].num_children;

            //set the right child as the new root
            subtree_root_index = right_child_index;
        }
        void rotate_right(size_t& subtree_root_index) {
            Node& subtree_root = nodes[subtree_root_index];
            size_t left_child_index = subtree_root.left_index;
            Node& left_child = nodes[left_child_index];

            //original root adopts the left child's right subtree
            subtree_root.left_index = left_child.right_index;
            //original root adopted a subtree (whose height did not change), so update its height
            subtree_root.update_height(nodes);

            //left child adopts original root and its children
            left_child.right_index = subtree_root_index;
            //left child (new root) adopted the original root (whose height has been updated), so update its height
            left_child.update_height(nodes);
            //since left child took the subtree root's place, it has the same number of children as the original root
            left_child.num_children = subtree_root.num_children;

            //root has new children, so update that counter (done after changing the left child's children counter
            //because that depends on the original root's counter)
            subtree_root.num_children = 0;
            if (subtree_root.left_index != 0)
                subtree_root.num_children += 1 + nodes[subtree_root.left_index].num_children;
            if (subtree_root.right_index != 0)
                subtree_root.num_children += 1 + nodes[subtree_root.right_index].num_children;

            //set the left child as the new root
            subtree_root_index = left_child_index;
        }
        int do_search(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 value) const
        {
            if (subtree_root_index == 0)
                //key not found
                nodes_visited *= -1;
            else {
                Node const& subtree_root = nodes[subtree_root_index];
                ++nodes_visited;
                switch (compare(key, subtree_root.key)) {
                case -1:
                    //key is less than subtree root key
                    nodes_visited = do_search(nodes_visited, subtree_root.left_index, key, value);
                    break;
                case 1:
                    //key is greater than subtree root key
                    nodes_visited = do_search(nodes_visited, subtree_root.right_index, key, value);
                    break;
                case 0:
                    //found key
                    value = subtree_root.value;
                    break;
                default:
                    throw std::domain_error("Unexpected compare() function return value");
                }
            }
            return nodes_visited;
        }
        void prepare_cluster_distribution(size_t subtree_root_index,
                                          size_t curr_height, //includes the height of the current node, ie assumes current node exists
                                          size_t cluster_counter[])
        {
            Node const& subtree_root = nodes[subtree_root_index];
            if ( ! subtree_root.left_index && ! subtree_root.right_index)
                //at a leaf node
                cluster_counter[curr_height]++;
            else {
                if (subtree_root.left_index)
                    prepare_cluster_distribution(subtree_root.left_index, curr_height + 1, cluster_counter);
                if (subtree_root.right_index)
                    prepare_cluster_distribution(subtree_root.right_index, curr_height + 1, cluster_counter);
            }
        }

        void remove_ith_node_inorder(size_t& subtree_root_index,
                                  size_t& ith_node_to_delete,
                                  key_type& key)
        {
            Node& subtree_root = nodes[subtree_root_index];
            if (subtree_root.left_index)
                remove_ith_node_inorder(subtree_root.left_index, ith_node_to_delete, key);
            if (ith_node_to_delete == 0)
                //deleted node in child subtree; nothing more to do
                return;
            if (--ith_node_to_delete == 0) {
                //delete the current node
                value_type dummy_val;
                remove(subtree_root.key, dummy_val);
                key = subtree_root.key;
                return;
            }
            if (subtree_root.right_index)
                remove_ith_node_inorder(subtree_root.right_index, ith_node_to_delete, key);
        }

    public:
        /*
            The constructor will allocate an array of capacity (binary
            tree) nodes. Then make a chain from all the nodes (e.g.,
            make node 2 the left child of node 1, make node 3 the left
            child of node 2, &c. this is the initial free list.
        */
        BST(size_t capacity):
            curr_capacity(capacity)
        {
            if (capacity == 0) {
                throw std::domain_error("capacity must be at least 1");
            }
            nodes = new Node[capacity + 1];
            clear();
        }
        /*
            if there is space available, adds the specified key/value-pair to the tree
            and returns the number of nodes visited, V; otherwise returns -1 * V. If an
            item already exists in the tree with the same key, replace its value.
        */
        virtual int insert(key_type const& key, value_type const& value) {
            if (size() == capacity())
                //no more space
                return 0;
            bool found_key = false;
            return insert_at_leaf(0, root_index, key, value, found_key);
        }
        /*
            if there is an item matching key, removes the key/value-pair from the tree, stores
            it’s value in value, and returns the number of probes required, V; otherwise returns -1 * V.
        */
        virtual int remove(key_type const& key, value_type& value) {
            bool found_key = false;
            return do_remove(0, root_index, key, value, found_key);
        }
        /*
            if there is an item matching key, stores it’s value in value, and returns the number
            of nodes visited, V; otherwise returns -1 * V. Regardless, the item remains in the tree.
        */
        virtual int search(key_type const& key, value_type& value) {
            return do_search(0, root_index, key, value);
        }
        /*
            removes all items from the map
        */
        virtual void clear() {
            //Since I use size_t to hold the node indices, I make the node array
            //1-based, with child index of 0 indicating that the current node is a leaf
            for (size_t i = 1; i != capacity(); ++i)
                nodes[i].disable_and_adopt_free_tree(i + 1);
            free_index = 1;
            root_index = 0;
        }
        /*
            returns true IFF the map contains no elements.
        */
        virtual bool is_empty() const {
            return size() == 0;
        }
        /*
            returns the number of slots in the backing array.
        */
        virtual size_t capacity() const {
            return curr_capacity;
        }
        /*
            returns the number of items actually stored in the tree.
        */
        virtual size_t size() const {
            if (root_index == 0) return 0;
            Node const& root = nodes[root_index];
            return 1 + root.num_children;
        }
        /*
            [not a regular BST operation, but specific to this implementation]
            returns the tree's load factor: load = size / capacity.
        */
        virtual double load() const {
            return static_cast<double>(size()) / capacity();
        }
        /*
            prints the tree in the following format:
            +--[tiger]
            |  |
            |  |  +--[panther]
            |  |  |
            |  +--[ocelot]
            |     |
            |     +--[lion]
            |
            [leopard]
            |
            |     +--[house cat]
            |     |
            |  +--[cougar]
            |  |
            +--[cheetah]
               |
               +--[bobcat]
        */
        virtual std::ostream& print(std::ostream& out) const {
            if (root_index == 0)
                return out;
            size_t num_lines = size() * 2 - 1;
            //use CDAL here so we can print really super-huge trees where the write buffer doesn't fit in memory
            CDAL<std::string> buffer_lines(100000);
            for(size_t i = 0; i <= num_lines; ++i)
                buffer_lines.push_back("");
            Node const& root = nodes[root_index];
            size_t root_line_index = 1;
            if (root.right_index) {
                root_line_index += 2 * (1 + nodes[root.right_index].num_children);
            }
            write_subtree_buffer(root_index, buffer_lines, root_line_index, 1, num_lines + 1);
            for (size_t i = 1; i <= num_lines; ++i)
                out << buffer_lines[i] << std::endl;
            return out;
        }

        /*
```

```
            returns a list indicating the number of leaf nodes at each height (since the RBST doesn’t exhibit
            true clustering, but can have degenerate branches).
        */
        virtual priority_queue<hash_utils::ClusterInventory> cluster_distribution() {
            //use an array to count cluster instances, then feed those to a priority queue and return it.
            priority_queue<ClusterInventory> cluster_pq;
            if (is_empty()) return cluster_pq;
            size_t max_height = nodes[root_index].height;
            size_t cluster_counter[max_height + 1];
            for (size_t i = 0; i <= max_height; ++i)
                cluster_counter[i] = 0;
            prepare_cluster_distribution(root_index, 1, cluster_counter);
            for (size_t i = 1; i <= max_height; ++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 the root (node 1), do an in-order
            traversal to find the R-th occupied node; remove that node (adjusting its children accordingly),
            and return its key.
        */
        virtual key_type remove_random() {
            if (size() == 0) throw std::logic_error("Cant remove from an empty map");
            size_t ith_node_to_delete = 1 + hash_utils::rand_i(size());
            key_type key;
            remove_ith_node_inorder(root_index, ith_node_to_delete, key);
            return key;
        }
    };
}

#endif
```

part4_bonus/source/avl.h

part4_bonus/source/avl.h

```cpp
#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);
            balance(subtree_root_index);
            return largest_key_node_index;
        }
        int do_remove(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& value,
                     bool& found_key)
        {
            nodes_visited = super::do_remove(nodes_visited, subtree_root_index, key, value, found_key);
            balance(subtree_root_index);
            return nodes_visited;
        }
        void balance(size_t& subtree_root_index) {
            if (subtree_root_index == 0) return;
            Node& root = this->nodes[subtree_root_index];
            int root_bal_fact = root.balance_factor(this->nodes);
            if (root_bal_fact == -2) {
                //right subtree is too heavy
                size_t& right_index = root.right_index;
                Node& right_child = this->nodes[right_index];
                switch(right_child.balance_factor(this->nodes)) {
                case 1:
                    //right left
                    this->rotate_right(right_index);
                    this->rotate_left(subtree_root_index);
                    break;
                case -1:
                case 0:
                    //right right
                    this->rotate_left(subtree_root_index);
                    break;
                default:
                    throw std::domain_error(std::string("Unexpected balance factor with heavy right subtree: ")
                                          + std::to_string(right_child.balance_factor(this->nodes)));
                }
            } else if (root_bal_fact == 2) {
                //left subtree is too heavy
                size_t& left_index = root.left_index;
                Node& left_child = this->nodes[left_index];
                switch(left_child.balance_factor(this->nodes)) {
                case -1:
                    //left right
                    this->rotate_left(left_index);
                    this->rotate_right(subtree_root_index);
                    break;
                case 1:
                case 0:
                    //left left
                    this->rotate_right(subtree_root_index);
                    break;
                default:
                    throw std::domain_error(std::string("Unexpected balance factor with heavy left subtree: ")
                                             + std::to_string(left_child.balance_factor(this->nodes)));
                }
            } else if (std::abs(root_bal_fact > 2)) {
                throw std::domain_error(std::string("Unexpected balance factor when checking for heavy subtree: ")
                                         + std::to_string(root_bal_fact));
            }
        }
        void do_validate_integrity(size_t subtree_root_index) const {
            if (subtree_root_index == 0) return;
            Node const& n = this->nodes[subtree_root_index];
            if (abs(n.balance_factor(this->nodes)) > 1)
                throw std::domain_error("Unexpected unbalanced tree while checking balance factor of all tree nodes");
            do_validate_integrity(n.left_index);
            do_validate_integrity(n.right_index);
        }
        void validate_integrity() {
            do_validate_integrity(this->root_index);
        }
    public:
        AVL(size_t capacity): super(capacity) {}
        /*
            if there is space available, adds the specified key/value-pair to the tree
            and returns the number of nodes visited, V; otherwise returns -1 * V. If an
            item already exists in the tree with the same key, replace its value.
        */
        int insert(key_type const& key, value_type const& value) {
            if (this->size() == this->capacity())
                //no more space
                return 0;
            bool found_key = false;
            return insert_at_leaf(0, this->root_index, key, value, found_key);
        }
        /*
            if there is an item matching key, removes the key/value-pair from the tree, stores
            it's value in value, and returns the number of probes required, V; otherwise returns -1 * V.
        */
        int remove(key_type const& key, value_type& value) {
            bool found_key = false;
            int nodes_visited = do_remove(0, this->root_index, key, value, found_key);
            validate_integrity();
            return nodes_visited;
        }
    };
}

#endif
```