Project 1 Deliverable

Paul Nickerson

November 24, 2014

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

SSLL

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

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

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

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->next==nullptr

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

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 reference 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->next==nullptr

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

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

PSLL

PSLL 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

PSLL()

- Default constructor - initializes the head, tail, and free-head dummy nodes

PSLL(const PSLL& 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

PSLL& operator=(const PSLL& 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

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 PSLL_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

PSLL_Iter(const PSLL_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

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->next==nullptr

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 PSLL_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

PSLL_Const_Iter(const PSLL_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

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 reference 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->next==nullptr

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

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

SDAL

SDAL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.
- Passes a pointer to the end slot so that the iterator can do bounds checking

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.
- Passes a pointer to the end slot so that the iterator can do bounds checking

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

SDAL(const SDAL& src)

- Copy constructor - starting from uninitialized state, initialize the class by allocating a number of nodes equal to the source instance's array size, then use an iterator to push_bash() each source item into the current list
- If we fail to allocate nodes, throw a bad_alloc exception
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

SDAL& operator=(const SDAL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state by freeing the item array, 2) initialize the class by allocating a number of nodes equal to the source instance's array size, and 3) use an iterator to push_bash() each source item into the current list
- If we fail to allocate nodes, throw a bad_alloc exception
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

void embiggen_if_necessary()

- Called whenever we attempt to increase the list size
- Checks if backing array is full, and if so, allocate a new array 150% the size of the original, copy the items over to the new array, and deallocate the original one.
- If we fail to allocate nodes, throw a bad_alloc exception

void shrink_if_necessary()

- Called whenever we attempt to decrease the list size

- Because we don't want the list to waste too much memory, whenever the array's size is >= 100 slots and fewer than half the slots are used, allocate a new array 50% the size of the original, copy the items over to the new array, and deallocate the original one.
- If we fail to allocate nodes, throw a bad_alloc exception

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."
- Calls embiggen_if_necessary() to ensure we have space to insert the new item
- 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
- Providing a position greater than the current list size should throw an out-of-range error

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 calling insert() with the position defined as one past the last stored item
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Wrapper for remove(0)
- Removes the node at item_array[0] and returns its stored item

- If the list is empty then throw an out-of-range error
- It would be an error if, after popping, size() returned anything besides the old value returned from size() minus one

T pop_back()

- Wrapper for remove(size() - 1)
- Removes last stored node, returning its item
- If the list is empty then throw an out-of-range error
- It would be an error if, after popping, size() returned anything besides the old value returned from size() minus one

T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left" by traversing from the specified slot to the end of the array and moving each item to its preceding slot
- May only be called with positions *less than* the current list size
- It would be an error if, after removing, size() returned anything besides the old value returned from size() minus one

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

void clear()

- Removes all elements in the list by setting the counter holding the list size to zero. No further action is taken as it is assumed that the embiggen/shrink methods will handle it

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 SDAL_Iter(T* item_array, T* end_ptr)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the first item held in the item_array parameter
- Neither item_array nor end_ptr may be null
- end_ptr must be greater than or equal to item_array

SDAL_Iter(const SDAL_Iter& src)

- Copy constructor - sets the current iterator position in the item array and the end 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

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 and end 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 iter==iter_end

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 current and end iter pointers match between current instance and rhs, otherwise returns false

bool operator!=(const self_type& rhs) const

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

Const Iterator Methods

explicit SDAL_Const_Iter(Node* start)

- Explicit constructor for an iterator which returns an immutable reference to the first item held in the item_array parameter
- Neither item_array nor end_ptr may be null
- end_ptr must be greater than or equal to item_array

SDAL_Const_Iter(const SDAL_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

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 reference 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 iter==iter_end

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 current and end iter pointers match between current instance and rhs, otherwise returns false

bool operator!=(const self_type& rhs) const

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

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

CDAL

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()

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()

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
- Decrements size by one
- 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 caled 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

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->next==nullptr

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

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 reference 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->next==nullptr

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

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

SSLL checklist & source code

ssll/checklist.txt

```
Simple, Singly Linked List written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part I:
=============================================================================
My LIST implementation uses the data structure described in the part I
instructions and conforms to the technique required for this list
variety: yes

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

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes

=============================================================================
Part II:
=============================================================================
My LIST implementation 100% correctly supports the following methods
as described in part II:

* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes

My LIST implementation 100% correctly supports the following data
members as described in part II:

* size_t
* value_type
* iterator
* const_iterator

My ITERATOR implementation 100% correctly supports the following
```

```
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

My CONST ITERATOR implementation 100% correctly supports the following
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My CONST ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

=========================================================================
```

```
Part III:
=========================================================================
My LIST implementation 100% correctly supports the following
methods as described in part III:

* operator[]: yes
* operator[] const: yes

For my LIST's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes
* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes
* operator[]: yes
* operator[] const: yes

For my ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*
```

```
- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

For my CONST ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

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

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

My UNIT 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 Simple, Singly Linked List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

=====================================================================
In addition to the unit tests, the old_tests directory contains a fuzzer
which stress-tests every list and compares their states to ensure they
all behave equivalently as well as maintain internal integrity.
=====================================================================

How to compile and run my unit tests on the OpenBSD VM
cd list_source_directory
./compile.sh
./unit_tester -s > output.txt
```

ssll/source/SSLL.h

SSLL.h

```
//note to self: global search for todo and xxx before turning this assignment in
#ifndef _SSLL_H_
#define _SSLL_H_

// SSLL.H
//
// Singly-linked list (non-polymorphic)
//
// Authors: Paul Nickerson, Dave Small
// for COP 3530
// 201409.16 - created

#include <iostream>
#include <stdexcept>
#include <cassert>

namespace cop3530 {
    template <class T>
    class SSLL {
    private:
        struct Node {
            T item;
            Node* next;
            bool is_dummy;
        }; // end struct Node
        size_t num_items;
        Node* head;
        Node* tail;
        Node* node_at(size_t position) const {
            Node* n = head->next;
            for (size_t i = 0; i != position; ++i, n = n->next);
            return n;
        }
        Node* node_before(size_t position) const {
            if (position == 0)
```

```
48              return head;
49          else
50              return node_at(position - 1);
51      }
52      Node* design_new_node(const T& element, Node* next = nullptr, bool dummy =
            false) {
53          Node* n = new Node();
54          n->is_dummy = dummy;
55          n->item = element;
56          n->next = next;
57          return n;
58      }
59      Node* design_new_node(Node* next = nullptr, bool dummy = false) {
60          Node* n = new Node();
61          n->is_dummy = dummy;
62          n->next = next;
63          return n;
64      }
65      void init() {
66          num_items = 0;
67          try {
68              tail = design_new_node(nullptr, true);
69              head = design_new_node(tail, true);
70          } catch (std::bad_alloc& ba) {
71              std::cerr << "init(): failed to allocate memory for head/tail nodes"
                        << std::endl;
72              throw std::bad_alloc();
73          }
74      }
75      //note to self: the key to simple ssll navigation is to frame the problem
            in terms of the following two functions (insert_node_after and
            remove_item_after)
76      void insert_node_after(Node* existing_node, Node* new_node) {
77          existing_node->next = new_node;
78          ++num_items;
79      }
80      //destroys the subsequent node and returns its item
81      T remove_item_after(Node* preceeding_node) {
82          Node* removed_node = preceeding_node->next;
83          T item = removed_node->item;
84          preceeding_node->next = removed_node->next;
85          delete removed_node;
86          --num_items;
87          return item;
88      }
89      void copy_constructor(const SSLL& src) {
90          const_iterator fin = src.end();
91          for (const_iterator iter = src.begin(); iter != fin; ++iter) {
92              push_back(*iter);
93          }
94          if ( ! src.size() == size())
```

```
95              throw std::runtime_error("copy_constructor: Copying failed - sizes
                    don't match up");
96          }
97      public:
98
99          //-------------------------------------------------
100         // iterators
101         //-------------------------------------------------
102         class SSLL_Iter: public std::iterator<std::forward_iterator_tag, T>
103         {
104         public:
105             // inheriting from std::iterator<std::forward_iterator_tag, T>
106             // automagically sets up these typedefs...
107             typedef T value_type;
108             typedef std::ptrdiff_t difference_type;
109             typedef T& reference;
110             typedef T* pointer;
111             typedef std::forward_iterator_tag iterator_category;
112
113             // but not these typedefs...
114             typedef SSLL_Iter self_type;
115             typedef SSLL_Iter& self_reference;
116
117         private:
118             Node* here;
119
120         public:
121             explicit SSLL_Iter(Node* start) : here(start) {
122                 if (start == nullptr)
123                     throw std::runtime_error("SSLL_Iter: start cannot be null");
124             }
125             SSLL_Iter(const SSLL_Iter& src) : here(src.here) {
126                 if (*this != src)
127                     throw std::runtime_error("SSLL_Iter: copy constructor failed");
128             }
129             reference operator*() const {
130                 return here->item;
131             }
132             pointer operator->() const {
133                 return & this->operator*();
134             }
135             self_reference operator=( const self_type& src ) {
136                 if (&src == this)
137                     return *this;
138                 here = src.here;
139                 if (*this != src)
140                     throw std::runtime_error("SSLL_Iter: copy assignment failed");
141                 return *this;
142             }
143             self_reference operator++() { // preincrement
144                 if (here->next == nullptr)
```

```
145                 throw std::out_of_range("SSLL_Iter: Can't traverse past the end
                        of the list");
146             here = here->next;
147             return *this;
148         }
149         self_type operator++(int) { // postincrement
150             self_type t(*this); //save state
151             operator++(); //apply increment
152             return t; //return state held before increment
153         }
154         bool operator==(const self_type& rhs) const {
155             return rhs.here == here;
156         }
157         bool operator!=(const self_type& rhs) const {
158             return ! operator==(rhs);
159         }
160     };
161
162     class SSLL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
163     {
164     public:
165         // inheriting from std::iterator<std::forward_iterator_tag, T>
166         // automagically sets up these typedefs...
167         typedef T value_type;
168         typedef std::ptrdiff_t difference_type;
169         typedef const T& reference;
170         typedef const T* pointer;
171         typedef std::forward_iterator_tag iterator_category;
172
173         // but not these typedefs...
174         typedef SSLL_Const_Iter self_type;
175         typedef SSLL_Const_Iter& self_reference;
176
177     private:
178         const Node* here;
179
180     public:
181         explicit SSLL_Const_Iter(Node* start) : here(start) {
182             if (start == nullptr)
183                 throw std::runtime_error("SSLL_Const_Iter: start cannot be null");
184         }
185         SSLL_Const_Iter(const SSLL_Const_Iter& src) : here(src.here) {
186             if (*this != src)
187                 throw std::runtime_error("SSLL_Const_Iter: copy constructor
                        failed");
188         }
189
190         reference operator*() const {
191             return here->item;
192         }
193         pointer operator->() const {
194             return & this->operator*();
```

```
195             }
196             self_reference operator=( const self_type& src ) {
197                 if (&src == this)
198                     return *this;
199                 here = src.here;
200                 if (*this != src)
201                     throw std::runtime_error("SSLL_Const_Iter: copy assignment
                        failed");
202                 return *this;
203             }
204             self_reference operator++() { // preincrement
205                 if (here->next == nullptr)
206                     throw std::out_of_range("SSLL_Const_Iter: Can't traverse past the
                        end of the list");
207                 here = here->next;
208                 return *this;
209             }
210             self_type operator++(int) { // postincrement
211                 self_type t(*this); //save state
212                 operator++(); //apply increment
213                 return t; //return state held before increment
214             }
215             bool operator==(const self_type& rhs) const {
216                 return rhs.here == here;
217             }
218             bool operator!=(const self_type& rhs) const {
219                 return ! operator==(rhs);
220             }
221         };
222
223         //-------------------------------------------------
224         // types
225         //-------------------------------------------------
226         typedef T value_type;
227         typedef SSLL_Iter iterator;
228         typedef SSLL_Const_Iter const_iterator;
229
230         iterator begin() { return SSLL_Iter(head->next); }
231         iterator end() { return SSLL_Iter(tail); }
232
233         const_iterator begin() const { return SSLL_Const_Iter(head->next); }
234         const_iterator end() const { return SSLL_Const_Iter(tail); }
235
236         //-------------------------------------------------
237         // operators
238         //-------------------------------------------------
239         T& operator[](size_t i) {
240             if (i >= size()) {
241                 throw std::out_of_range(std::string("operator[]: No element at
                    position ") + std::to_string(i));
242             }
243             return node_at(i)->item;
```

```
244         }
245
246         const T& operator[](size_t i) const {
247             if (i >= size()) {
248                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
249             }
250             return node_at(i)->item;
251         }
252
253         //-------------------------------------------------
254         // Constructors/destructor/assignment operator
255         //-------------------------------------------------
256
257         SSLL() {
258             init();
259         }
260         //-------------------------------------------------
261         //copy constructor
262         //note to self: src must be const in case we want to assign this from a
                const source
263         SSLL(const SSLL& src) {
264             init();
265             copy_constructor(src);
266         }
267
268         //-------------------------------------------------
269         //destructor
270         ~SSLL() {
271             // safely dispose of this SSLL's contents
272             clear();
273         }
274
275         //-------------------------------------------------
276         //copy assignment constructor
277         SSLL& operator=(const SSLL& src) {
278             if (&src == this) // check for self-assignment
279                 return *this;   // do nothing
280             // safely dispose of this SSLL's contents
281             clear();
282             // populate this SSLL with copies of the other SSLL's contents
283             copy_constructor(src);
284             return *this;
285         }
286
287         //-------------------------------------------------
288         // member functions
289         //-------------------------------------------------
290
291         /*
292             replaces the existing element at the specified position with the
                    specified element and
```

```
293             returns the original element.
294         */
295         T replace(const T& element, size_t position) {
296             T old_item;
297             if (position >= size()) {
298                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
299             } else {
300                 //we are guaranteed to be at a non-dummy item now because of the
                        above if statement
301                 Node* iter = node_at(position);
302                 old_item = iter->item;
303                 iter->item = element;
304             }
305             return old_item;
306         }
307 
308         //-------------------------------------------------
309         /*
310             adds the specified element to the list at the specified position,
                    shifting the element
311             originally at that and those in subsequent positions one position to the
                    right.
312         */
313         void insert(const T& element, size_t position) {
314             if (position > size()) {
315                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
316             } else if (position == size()) {
317                 //special O(1) case
318                 push_back(element);
319             } else {
320                 //node_before_position is guaranteed to point to a valid node
                        because we use a dummy head node
321                 Node* node_before_position = node_before(position);
322                 Node* node_at_position = node_before_position->next;
323                 Node* new_node;
324                 try {
325                     new_node = design_new_node(element, node_at_position);
326                 } catch (std::bad_alloc& ba) {
327                     std::cerr << "insert(): failed to allocate memory for new node"
                            << std::endl;
328                     throw std::bad_alloc();
329                 }
330                 insert_node_after(node_before_position, new_node);
331             }
332         }
333 
334         /*
335             prepends the specified element to the list.
336         */
337         void push_front(const T& element) {
```

```
338             insert(element, 0);
339         }
340 
341         //-------------------------------------------------
342         /*
343             appends the specified element to the list.
344         */
345         void push_back(const T& element) {
346             Node* new_tail;
347             try {
348                 new_tail = design_new_node(nullptr, true);
349             } catch (std::bad_alloc& ba) {
350                 std::cerr << "push_back(): failed to allocate memory for new tail"
                        << std::endl;
351                 throw std::bad_alloc();
352             }
353             insert_node_after(tail, new_tail);
354             //transform the current tail node from a dummy to a real node holding
                    element
355             tail->is_dummy = false;
356             tail->item = element;
357             tail->next = new_tail;
358             tail = tail->next;
359         }
360 
361         /*
362             removes and returns the element at the list's head.
363         */
364         T pop_front() {
365             if (is_empty()) {
366                 throw std::out_of_range("pop_front: Can't pop: list is empty");
367             }
368             if (head->next == tail) {
369                 throw std::runtime_error("pop_front: head->next == tail, but list
                        says it's not empty (corrupt state)");
370             }
371             return remove_item_after(head);
372         }
373 
374         //-------------------------------------------------
375         /*
376             removes and returns the element at the list's tail.
377         */
378         T pop_back() {
379             if (is_empty()) {
380                 throw std::out_of_range("pop_back: Can't pop: list is empty");
381             }
382             if (head->next == tail) {
383                 throw std::runtime_error("pop_back: head->next == tail, but list
                        says it's not empty (corrupt state)");
384             }
385             //XXX this is O(N), a disadvantage of this architecture
```

```
386             Node* node_before_last = node_before(size() - 1);
387             T item = remove_item_after(node_before_last);
388             return item;
389         }
390 
391         //-------------------------------------------------
392         /*
393             removes and returns the the element at the specified position,
394             shifting the subsequent elements one position to the left.
395         */
396         T remove(size_t position) {
397             T item;
398             if (position >= size()) {
399                 throw std::out_of_range(std::string("remove: No element at position
                        ") + std::to_string(position));
400             }
401             if (head->next == tail) {
402                 throw std::runtime_error("remove: head->next == tail, but list says
                        it's not empty (corrupt state)");
403             }
404             //using a dummy head node guarantees that there be a node immediately
                    preceeding the specified position
405             Node *node_before_position = node_before(position);
406             item = remove_item_after(node_before_position);
407             return item;
408         }
409 
410         //-------------------------------------------------
411         /*
412             returns (without removing from the list) the element at the specified
                    position.
413         */
414         T item_at(size_t position) const {
415             if (position >= size()) {
416                 throw std::out_of_range(std::string("item_at: No element at position
                        ") + std::to_string(position));
417             }
418             return operator[](position);
419         }
420 
421         //-------------------------------------------------
422         /*
423             returns true IFF the list contains no elements.
424         */
425         bool is_empty() const {
426             return size() == 0;
427         }
428 
429         //-------------------------------------------------
430         /*
431             returns the number of elements in the list.
432         */
```

```
433         size_t size() const {
434             if (num_items == 0 && head->next != tail) {
435                 throw std::runtime_error("size: head->next != tail, but list says
                        it's empty (corrupt state)");
436             } else if (num_items > 0 && head->next == tail) {
437                 throw std::runtime_error("size: head->next == tail, but list says
                        it's not empty (corrupt state)");
438             }
439             return num_items;
440         }
441 
442         //-------------------------------------------------
443         /*
444             removes all elements from the list.
445         */
446         void clear() {
447             while ( ! is_empty()) {
448                 pop_front();
449             }
450         }
451 
452         //-------------------------------------------------
453         /*
454             returns true IFF one of the elements of the list matches the specified
                 element.
455         */
456         bool contains(const T& element,
457                bool equals(const T& a, const T& b)) const {
458             bool element_in_list = false;
459             const_iterator fin = end();
460             for (const_iterator iter = begin(); iter != fin; ++iter) {
461                 if (equals(*iter, element)) {
462                     element_in_list = true;
463                     break;
464                 }
465             }
466             return element_in_list;
467         }
468 
469         //-------------------------------------------------
470         /*
471             If the list is empty, inserts "<empty list>" into the ostream;
472             otherwise, inserts, enclosed in square brackets, the list's elements,
473             separated by commas, in sequential order.
474         */
475         std::ostream& print(std::ostream& out) const {
476             if (is_empty()) {
477                 out << "<empty list>";
478             } else {
479                 out << "[";
480                 const_iterator start = begin();
481                 const_iterator fin = end();
```

```
482                 for (const_iterator iter = start; iter != fin; ++iter) {
483                     if (iter != start)
484                         out << ",";
485                     out << *iter;
486                 }
487                 out << "]";
488             }
489             return out;
490         }
491     protected:
492         bool validate_internal_integrity() {
493             //todo: fill this in
494             return true;
495         }
496     }; //end class SSLL
497 } // end namespace cop3530
498 #endif // _SSLL_H_
```

PSLL checklist & source code

psll/checklist.txt

```
Pool-using Singly-Linked List written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part I:
=============================================================================
My LIST implementation uses the data structure described in the part I
instructions and conforms to the technique required for this list
variety: yes

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

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes

=============================================================================
Part II:
=============================================================================
My LIST implementation 100% correctly supports the following methods
as described in part II:

* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes

My LIST implementation 100% correctly supports the following data
members as described in part II:

* size_t
* value_type
* iterator
* const_iterator

My ITERATOR implementation 100% correctly supports the following
```

```
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

My CONST ITERATOR implementation 100% correctly supports the following
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My CONST ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

=====================================================================
```

```
Part III:
=========================================================================
My LIST implementation 100% correctly supports the following
methods as described in part III:

* operator[]: yes
* operator[] const: yes

For my LIST's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes
* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes
* operator[]: yes
* operator[] const: yes

For my ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*
```

```
- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

For my CONST ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

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

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

My UNIT 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 Pool-using Singly-Linked List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

=====================================================================
In addition to the unit tests, the old_tests directory contains a fuzzer
which stress-tests every list and compares their states to ensure they
all behave equivalently as well as maintain internal integrity.
=====================================================================

How to compile and run my unit tests on the OpenBSD VM
cd list_source_directory
./compile.sh
./unit_tester -s > output.txt
```

psll/source/PSLL.h

PSLL.h

```
#ifndef _PSLL_H_
#define _PSLL_H_

// PSLL.H
//
// Pool-using Singly-linked list (non-polymorphic)
//
// Authors: Paul Nickerson, Dave Small
// for COP 3530
// 201409.16 - created

#include <iostream>
#include <stdexcept>
#include <cassert>
#include <string>

namespace cop3530 {
    template <class T>
    class PSLL {
    private:
        struct Node {
            T item;
            Node* next;
            bool is_dummy;
        }; // end struct Node
        size_t num_main_list_items;
        size_t num_free_list_items;
        Node* head;
        Node* tail;
        Node* free_list_head;
        Node* node_at(size_t position) const {
            Node* n = head->next;
            for (size_t i = 0; i != position; ++i, n = n->next);
            return n;
        }
        Node* node_before(size_t position) const {
            if (position == 0)
                return head;
            else
                return node_at(position - 1);
        }
        Node* procure_free_node(bool force_allocation) {
            Node* n;
            if (force_allocation || free_list_size() == 0) {
                try {
                    n = new Node();
                } catch (std::bad_alloc& ba) {
```

```
48                  std::cerr << "procure_free_node(): failed to allocate new node"
                        << std::endl;
49                  throw std::bad_alloc();
50              }
51          } else {
52              n = remove_node_after(free_list_head, num_free_list_items);
53          }
54          return n;
55      }
56      void shrink_pool_if_necessary() {
57          if (size() >= 100) {
58              size_t old_size = size();
59              while (free_list_size() > size() / 2) { //while the pool contains
                    more nodes than half the list size
60                  Node* n = remove_node_after(free_list_head, num_free_list_items);
61                  delete n;
62              }
63              if (size() != old_size / 2) {
64                  throw std::runtime_error("shrink_pool_if_necessary: incorrect
                        resulting pool size");
65              }
66          }
67      }
68
69      size_t free_list_size() { return num_free_list_items; }
70      Node* design_new_node(const T& element, Node* next = nullptr, bool dummy =
            false, bool force_allocation = false) {
71          Node* n = procure_free_node(force_allocation);
72          n->is_dummy = dummy;
73          n->item = element;
74          n->next = next;
75          return n;
76      }
77      Node* design_new_node(Node* next = nullptr, bool dummy = false, bool
            force_allocation = false) {
78          Node* n = procure_free_node(force_allocation);
79          n->is_dummy = dummy;
80          n->next = next;
81          return n;
82      }
83      void init() {
84          num_main_list_items = 0;
85          num_free_list_items = 0;
86          free_list_head = design_new_node(nullptr, true, true);
87          tail = design_new_node(nullptr, true, true);
88          head = design_new_node(tail, true, true);
89      }
90      void copy_constructor(const PSLL& src) {
91          //note: this function does *not* copy the free list
92          const_iterator fin = src.end();
93          for (const_iterator iter = src.begin(); iter != fin; ++iter) {
94              push_back(*iter);
```

```
95              }
96              if ( ! src.size() == size())
97                  throw std::runtime_error("copy_constructor: Copying failed - sizes
                        don't match up");
98          }
99          Node* remove_node_after(Node* preceeding_node, size_t& list_size_counter) {
100             if (preceeding_node->next == tail) {
101                 throw std::runtime_error("remove_node_after:
                        preceeding_node->next==tail, and we cant remove the tail");
102             }
103             if (preceeding_node == tail) {
104                 throw std::runtime_error("remove_node_after: preceeding_node==tail,
                        and we cant remove after the tail");
105             }
106             if (preceeding_node == free_list_head && free_list_size() == 0) {
107                 throw std::runtime_error("remove_node_after: attempt detected to
                        remove a node from an empty pool");
108             }
109             Node* removed_node = preceeding_node->next;
110             preceeding_node->next = removed_node->next;
111             removed_node->next = nullptr;
112             --list_size_counter;
113             return removed_node;
114         }
115
116         void insert_node_after(Node* existing_node, Node* new_node, size_t&
                list_size_counter) {
117             new_node->next = existing_node->next;
118             existing_node->next = new_node;
119             ++list_size_counter;
120         }
121
122         //returns subsequent node's item and moves that node to the free pool
123         T remove_item_after(Node* preceeding_node) {
124             Node* removed_node = remove_node_after(preceeding_node,
                        num_main_list_items);
125             T item = removed_node->item;
126             insert_node_after(free_list_head, removed_node, num_free_list_items);
127             shrink_pool_if_necessary();
128             return item;
129         }
130
131     public:
132         //-------------------------------------------------
133         // iterators
134         //-------------------------------------------------
135         class PSLL_Iter: public std::iterator<std::forward_iterator_tag, T>
136         {
137         public:
138             // inheriting from std::iterator<std::forward_iterator_tag, T>
139             // automagically sets up these typedefs...
140             typedef T value_type;
```

```
141             typedef std::ptrdiff_t difference_type;
142             typedef T& reference;
143             typedef T* pointer;
144             typedef std::forward_iterator_tag iterator_category;
145 
146             // but not these typedefs...
147             typedef PSLL_Iter self_type;
148             typedef PSLL_Iter& self_reference;
149 
150         private:
151             Node* here;
152 
153         public:
154             explicit PSLL_Iter(Node* start) : here(start) {
155                 if (start == nullptr)
156                     throw std::runtime_error("PSLL_Iter: start cannot be null");
157             }
158             PSLL_Iter(const PSLL_Iter& src) : here(src.here) {
159                 if (*this != src)
160                     throw std::runtime_error("PSLL_Iter: copy constructor failed");
161             }
162             reference operator*() const {
163                 return here->item;
164             }
165             pointer operator->() const {
166                 return & this->operator*();
167             }
168             self_reference operator=( const self_type& src ) {
169                 if (&src == this)
170                     return *this;
171                 here = src.here;
172                 if (*this != src)
173                     throw std::runtime_error("PSLL_Iter: copy assignment failed");
174                 return *this;
175             }
176             self_reference operator++() { // preincrement
177                 if (here->next == nullptr)
178                     throw std::out_of_range("PSLL_Iter: Can't traverse past the end
                        of the list");
179                 here = here->next;
180                 return *this;
181             }
182             self_type operator++(int) { // postincrement
183                 self_type t(*this); //save state
184                 operator++(); //apply increment
185                 return t; //return state held before increment
186             }
187             bool operator==(const self_type& rhs) const {
188                 return rhs.here == here;
189             }
190             bool operator!=(const self_type& rhs) const {
191                 return ! operator==(rhs);
```

```
192             }
193         };
194
195         class PSLL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
196         {
197         public:
198             // inheriting from std::iterator<std::forward_iterator_tag, T>
199             // automagically sets up these typedefs...
200             typedef T value_type;
201             typedef std::ptrdiff_t difference_type;
202             typedef const T& reference;
203             typedef const T* pointer;
204             typedef std::forward_iterator_tag iterator_category;
205
206             // but not these typedefs...
207             typedef PSLL_Const_Iter self_type;
208             typedef PSLL_Const_Iter& self_reference;
209
210         private:
211             const Node* here;
212
213         public:
214             explicit PSLL_Const_Iter(Node* start) : here(start) {
215                 if (start == nullptr)
216                     throw std::runtime_error("PSLL_Const_Iter: start cannot be null");
217             }
218             PSLL_Const_Iter(const PSLL_Const_Iter& src) : here(src.here) {
219                 if (*this != src)
220                     throw std::runtime_error("PSLL_Const_Iter: copy constructor
                            failed");
221             }
222
223             reference operator*() const {
224                 return here->item;
225             }
226             pointer operator->() const {
227                 return & this->operator*();
228             }
229             self_reference operator=( const self_type& src ) {
230                 if (&src == this)
231                     return *this;
232                 here = src.here;
233                 if (*this != src)
234                     throw std::runtime_error("PSLL_Const_Iter: copy assignment
                            failed");
235                 return *this;
236             }
237             self_reference operator++() { // preincrement
238                 if (here->next == nullptr)
239                     throw std::out_of_range("PSLL_Const_Iter: Can't traverse past the
                            end of the list");
240                 here = here->next;
```

```
241             return *this;
242         }
243         self_type operator++(int) { // postincrement
244             self_type t(*this); //save state
245             operator++(); //apply increment
246             return t; //return state held before increment
247         }
248         bool operator==(const self_type& rhs) const {
249             return rhs.here == here;
250         }
251         bool operator!=(const self_type& rhs) const {
252             return ! operator==(rhs);
253         }
254     };
255 
256     //-------------------------------------------------
257     // types
258     //-------------------------------------------------
259     /*typedef std::size_t size_t;*/
260     typedef T value_type;
261     typedef PSLL_Iter iterator;
262     typedef PSLL_Const_Iter const_iterator;
263 
264     iterator begin() {
265         return iterator(head->next);
266     }
267     iterator end() {
268         return iterator(tail);
269     }
270     /*
271         Note to self: the following overloads will fail if not defined as const
272     */
273     const_iterator begin() const {
274         return const_iterator(head->next);
275     }
276     const_iterator end() const {
277         return const_iterator(tail);
278     }
279 
280     //-------------------------------------------------
281     // operators
282     //-------------------------------------------------
283     T& operator[](size_t i) {
284         if (i >= size()) {
285             throw std::out_of_range(std::string("operator[]: No element at
                    position ") + std::to_string(i));
286         }
287         return node_at(i)->item;
288     }
289 
290     const T& operator[](size_t i) const {
291         if (i >= size()) {
```

```
292             throw std::out_of_range(std::string("operator[]: No element at
                    position ") + std::to_string(i));
293         }
294         return node_at(i)->item;
295     }
296 
297     //------------------------------------------------
298     // Constructors/destructor/assignment operator
299     //------------------------------------------------
300 
301     PSLL() {
302         init();
303     }
304     //------------------------------------------------
305     //copy constructor
306     PSLL(const PSLL& src) {
307         init();
308         copy_constructor(src);
309     }
310 
311     //------------------------------------------------
312     //destructor
313     ~PSLL() {
314         // safely dispose of this PSLL's contents
315         clear();
316     }
317 
318     //------------------------------------------------
319     //copy assignment constructor
320     PSLL& operator=(const PSLL& src) {
321         if (&src == this) // check for self-assignment
322             return *this;   // do nothing
323         // safely dispose of this PSLL's contents
324         clear();
325         // populate this PSLL with copies of the other PSLL's contents
326         copy_constructor(src);
327         return *this;
328     }
329 
330     //------------------------------------------------
331     // member functions
332     //------------------------------------------------
333 
334     /*
335         replaces the existing element at the specified position with the
                specified element and
336         returns the original element.
337     */
338     T replace(const T& element, size_t position) {
339         T old_item;
340         if (position >= size()) {
```

```
341             throw std::out_of_range(std::string("replace: No element at position
                    ") + std::to_string(position));
342         } else {
343             //we are guaranteed to be at a non-dummy item now because of the
                    above if statement
344             Node* iter = node_at(position);
345             old_item = iter->item;
346             iter->item = element;
347         }
348         return old_item;
349     }
350 
351     //-------------------------------------------------
352     /*
353         adds the specified element to the list at the specified position,
                shifting the element
354         originally at that and those in subsequent positions one position to the
                right.
355     */
356     void insert(const T& element, size_t position) {
357         if (position > size()) {
358             throw std::out_of_range(std::string("insert: Position is outside of
                    the list: ") + std::to_string(position));
359         } else if (position == size()) {
360             //special O(1) case
361             push_back(element);
362         } else {
363             //node_before_position is guaranteed to point to a valid node
                    because we use a dummy head node
364             Node* node_before_position = node_before(position);
365             Node* node_at_position = node_before_position->next;
366             Node* new_node;
367             try {
368                 new_node = design_new_node(element, node_at_position);
369             } catch (std::bad_alloc& ba) {
370                 std::cerr << "insert(): failed to allocate memory for new node"
                        << std::endl;
371                 throw std::bad_alloc();
372             }
373             insert_node_after(node_before_position, new_node,
                    num_main_list_items);
374         }
375     }
376 
377     //-------------------------------------------------
378     //Note to self: use reference here because we receive the original object
            instance,
379     //then copy it into n->item so we have it if the original element goes out
            of scope
380     /*
381         prepends the specified element to the list.
382     */
```

```
383         void push_front(const T& element) {
384             insert(element, 0);
385         }
386 
387         //-------------------------------------------------
388         /*
389             appends the specified element to the list.
390         */
391         void push_back(const T& element) {
392             Node* new_tail;
393             try {
394                 new_tail = design_new_node(nullptr, true);
395             } catch (std::bad_alloc& ba) {
396                 std::cerr << "push_back(): failed to allocate memory for new tail"
                        << std::endl;
397                 throw std::bad_alloc();
398             }
399             insert_node_after(tail, new_tail, num_main_list_items);
400             //transform the current tail node from a dummy to a real node holding
                    element
401             tail->is_dummy = false;
402             tail->item = element;
403             tail->next = new_tail;
404             tail = tail->next;
405         }
406 
407         //-------------------------------------------------
408         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
409         //of that so the client still has a valid instance if our destructor is
                called
410         /*
411             removes and returns the element at the list's head.
412         */
413         T pop_front() {
414             if (is_empty()) {
415                 throw std::out_of_range("pop_front: Can't pop: list is empty");
416             }
417             if (head->next == tail) {
418                 throw std::runtime_error("pop_front: head->next == tail, but list
                        says it's not empty (corrupt state)");
419             }
420             return remove_item_after(head);
421         }
422 
423         //-------------------------------------------------
424         /*
425             removes and returns the element at the list's tail.
426         */
427         T pop_back() {
428             if (is_empty()) {
429                 throw std::out_of_range("pop_back: Can't pop: list is empty");
```

```
430             }
431             if (head->next == tail) {
432                 throw std::runtime_error("pop_back: head->next == tail, but list
                        says it's not empty (corrupt state)");
433             }
434             //XXX this is O(N), a disadvantage of this architecture
435             Node* node_before_last = node_before(size() - 1);
436             T item = remove_item_after(node_before_last);
437             return item;
438         }
439 
440         //-------------------------------------------------
441         /*
442             removes and returns the the element at the specified position,
443             shifting the subsequent elements one position to the left.
444         */
445         T remove(size_t position) {
446             T item;
447             if (position >= size()) {
448                 throw std::out_of_range(std::string("remove: No element at position
                        ") + std::to_string(position));
449             }
450             if (head->next == tail) {
451                 throw std::runtime_error("remove: head->next == tail, but list says
                        it's not empty (corrupt state)");
452             }
453             //using a dummy head node guarantees that there be a node immediately
                    preceeding the specified position
454             Node *node_before_position = node_before(position);
455             item = remove_item_after(node_before_position);
456             return item;
457         }
458         //-------------------------------------------------
459         /*
460             returns (without removing from the list) the element at the specified
                    position.
461         */
462         T item_at(size_t position) const {
463             if (position >= size()) {
464                 throw std::out_of_range(std::string("item_at: No element at position
                        ") + std::to_string(position));
465             }
466             return operator[](position);
467         }
468 
469 
470         //-------------------------------------------------
471         /*
472             returns true IFF the list contains no elements.
473         */
474         bool is_empty() const {
475             return size() == 0;
```

```
476         }
477
478         //-------------------------------------------------
479         /*
480             returns the number of elements in the list.
481         */
482         size_t size() const {
483             if (num_main_list_items == 0 && head->next != tail) {
484                 throw std::runtime_error("size: head->next != tail, but list says
                        it's empty (corrupt state)");
485             } else if (num_main_list_items > 0 && head->next == tail) {
486                 throw std::runtime_error("size: head->next == tail, but list says
                        it's not empty (corrupt state)");
487             }
488             return num_main_list_items;
489         }
490
491         //-------------------------------------------------
492         /*
493             removes all elements from the list.
494         */
495         void clear() {
496             while (size()) {
497                 pop_front();
498             }
499         }
500         //-------------------------------------------------
501         /*
502             returns true IFF one of the elements of the list matches the specified
                element.
503         */
504         bool contains(const T& element,
505               bool equals(const T& a, const T& b)) const {
506             bool element_in_list = false;
507             const_iterator fin = end();
508             for (const_iterator iter = begin(); iter != fin; ++iter) {
509                 if (equals(*iter, element)) {
510                     element_in_list = true;
511                     break;
512                 }
513             }
514             return element_in_list;
515         }
516
517         //-------------------------------------------------
518         /*
519             If the list is empty, inserts "<empty list>" into the ostream;
520             otherwise, inserts, enclosed in square brackets, the list's elements,
521             separated by commas, in sequential order.
522         */
523         std::ostream& print(std::ostream& out) const {
524             if (is_empty()) {
```

```
525                 out << "<empty list>";
526             } else {
527                 out << "[";
528                 const_iterator start = begin();
529                 const_iterator fin = end();
530                 for (const_iterator iter = start; iter != fin; ++iter) {
531                     if (iter != start)
532                         out << ",";
533                     out << *iter;
534                 }
535                 out << "]";
536             }
537             return out;
538         }
539     protected:
540         bool validate_internal_integrity() {
541             //todo: fill this in
542             return true;
543         }
544     }; //end class PSLL
545 } // end namespace cop3530
546 #endif // _PSLL_H_
```

SDAL checklist & source code

sdal/checklist.txt

```
Simple Dynamic Array-based List written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part I:
=============================================================================
My LIST implementation uses the data structure described in the part I
instructions and conforms to the technique required for this list
variety: yes

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

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes

=============================================================================
Part II:
=============================================================================
My LIST implementation 100% correctly supports the following methods
as described in part II:

* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes

My LIST implementation 100% correctly supports the following data
members as described in part II:

* size_t
* value_type
* iterator
* const_iterator

My ITERATOR implementation 100% correctly supports the following
```

```
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

My CONST ITERATOR implementation 100% correctly supports the following
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My CONST ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

======================================================================
```

```
Part III:
=========================================================================
My LIST implementation 100% correctly supports the following
methods as described in part III:

* operator[]: yes
* operator[] const: yes

For my LIST's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes
* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes
* operator[]: yes
* operator[] const: yes

For my ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*
```

```
- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

For my CONST ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

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

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

My UNIT 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 Simple Dynamic Array-based List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

=====================================================================
In addition to the unit tests, the old_tests directory contains a fuzzer
which stress-tests every list and compares their states to ensure they
all behave equivalently as well as maintain internal integrity.
=====================================================================

How to compile and run my unit tests on the OpenBSD VM
cd list_source_directory
./compile.sh
./unit_tester -s > output.txt
```

sdal/source/SDAL.h

SDAL.h

```
#ifndef _SDAL_H_
#define _SDAL_H_

// SDAL.H
//
// Singly-linked list (non-polymorphic)
//
// Authors: Paul Nickerson, Dave Small
// for COP 3530
// 201409.16 - created

#include <iostream>
#include <stdexcept>
#include <cassert>
#include <memory>
#include <string>
#include <cmath>

namespace cop3530 {
    template <class T>
    class SDAL {
    private:
        T* item_array;
        //XXX: do these both need to be size_t?
        size_t array_size;
        size_t num_items;
        size_t embiggen_counter = 0;
        size_t shrink_counter = 0;
        T* allocate_nodes(size_t quantity) {
            try {
                T* new_item_array = new T[quantity];
                return new_item_array;
            } catch (std::bad_alloc& ba) {
                std::cerr << "allocate_nodes(): failed to allocate item array of
                    size " << quantity << std::endl;
                throw std::bad_alloc();
            }
        }
        void embiggen_if_necessary() {
            /*
                Whenever an item is added and the backing array is full, allocate a
                    new array 150% the size
                of the original, copy the items over to the new array, and
                    deallocate the original one.
            */
            size_t filled_slots = size();
            if (filled_slots == array_size) {
```

```
45              size_t new_array_size = ceil(array_size * 1.5);
46              T* new_item_array = allocate_nodes(new_array_size);
47              for (size_t i = 0; i != filled_slots; ++i) {
48                  new_item_array[i] = item_array[i];
49              }
50              delete[] item_array;
51              item_array = new_item_array;
52              array_size = new_array_size;
53              ++embiggen_counter;
54          }
55      }
56      void shrink_if_necessary() {
57          /*
58              Because we don't want the list to waste too much memory, whenever
                    the array's size is  100 slots
59              and fewer than half the slots are used, allocate a new array 50% the
                    size of the original, copy
60              the items over to the new array, and deallocate the original one.
61          */
62          size_t filled_slots = size();
63          if (array_size >= 100 && filled_slots < array_size / 2) {
64              size_t new_array_size = ceil(array_size * 0.5);
65              T* new_item_array = allocate_nodes(new_array_size);
66              for (size_t i = 0; i != filled_slots; ++i) {
67                  new_item_array[i] = item_array[i];
68              }
69              delete[] item_array;
70              item_array = new_item_array;
71              array_size = new_array_size;
72              ++shrink_counter;
73          }
74      }
75      void init(size_t num_nodes_to_preallocate) {
76          array_size = num_nodes_to_preallocate;
77          num_items = 0;
78          item_array = allocate_nodes(array_size);
79      }
80      void copy_constructor(const SDAL& src) {
81          const_iterator fin = src.end();
82          for (const_iterator iter = src.begin(); iter != fin; ++iter) {
83              push_back(*iter);
84          }
85          if ( ! src.size() == size())
86              throw std::runtime_error("copy_constructor: Copying failed - sizes
                    don't match up");
87      }
88  public:
89
90      //-------------------------------------------------
91      // iterators
92      //-------------------------------------------------
93      class SDAL_Iter: public std::iterator<std::forward_iterator_tag, T>
```

```
94          {
95          public:
96              // inheriting from std::iterator<std::forward_iterator_tag, T>
97              // automagically sets up these typedefs...
98              //todo: figure out why we cant comment these out, which we should be
                    able to if they were
99              //defined when inheriting
100             typedef T value_type;
101             typedef std::ptrdiff_t difference_type;
102             typedef T& reference;
103             typedef T* pointer;
104             typedef std::forward_iterator_tag iterator_category;
105
106             // but not these typedefs...
107             typedef SDAL_Iter self_type;
108             typedef SDAL_Iter& self_reference;
109
110         private:
111             T* iter;
112             T* end_iter;
113
114         public:
115             explicit SDAL_Iter(T* item_array, T* end_ptr): iter(item_array),
                    end_iter(end_ptr) {
116                 if (item_array == nullptr)
117                     throw std::runtime_error("SDAL_Iter: item_array cannot be null");
118                 if (end_ptr == nullptr)
119                     throw std::runtime_error("SDAL_Iter: end_ptr cannot be null");
120                 if (item_array > end_ptr)
121                     throw std::runtime_error("SDAL_Iter: item_array pointer cannot be
                            past end_ptr");
122             }
123             SDAL_Iter(const SDAL_Iter& src): iter(src.iter), end_iter(src.end_iter) {
124                 if (*this != src)
125                     throw std::runtime_error("SDAL_Iter: copy constructor failed");
126             }
127             reference operator*() const {
128                 return *iter;
129             }
130             pointer operator->() const {
131                 return & this->operator*();
132             }
133             self_reference operator=( const self_type& src ) {
134                 if (&src == this)
135                     return *this;
136                 iter = src.iter;
137                 end_iter = src.end_iter;
138                 if (*this != src)
139                     throw std::runtime_error("SDAL_Iter: copy assignment failed");
140                 return *this;
141             }
142             self_reference operator++() { // preincrement
```

```
143             if (iter == end_iter)
144                 throw std::out_of_range("SDAL_Iter: Can't traverse past the end
                        of the list");
145             ++iter;
146             return *this;
147         }
148         self_type operator++(int) { // postincrement
149             self_type t(*this); //save state
150             operator++(); //apply increment
151             return t; //return state held before increment
152         }
153         bool operator==(const self_type& rhs) const {
154             return rhs.iter == iter && rhs.end_iter == end_iter;
155         }
156         bool operator!=(const self_type& rhs) const {
157             return ! operator==(rhs);
158         }
159     };
160
161     class SDAL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
162     {
163     public:
164         // inheriting from std::iterator<std::forward_iterator_tag, T>
165         // automagically sets up these typedefs...
166         typedef T value_type;
167         typedef std::ptrdiff_t difference_type;
168         typedef const T& reference;
169         typedef const T* pointer;
170         typedef std::forward_iterator_tag iterator_category;
171
172         // but not these typedefs...
173         typedef SDAL_Const_Iter self_type;
174         typedef SDAL_Const_Iter& self_reference;
175     private:
176         T* iter;
177         T* end_iter;
178     public:
179         explicit SDAL_Const_Iter(T* item_array, T* end_ptr): iter(item_array),
                end_iter(end_ptr) {
180             if (item_array == nullptr)
181                 throw std::runtime_error("SDAL_Const_Iter: item_array cannot be
                        null");
182             if (end_ptr == nullptr)
183                 throw std::runtime_error("SDAL_Const_Iter: end_ptr cannot be
                        null");
184             if (item_array > end_ptr)
185                 throw std::runtime_error("SDAL_Const_Iter: item_array pointer
                        cannot be past end_ptr");
186         }
187         SDAL_Const_Iter(const SDAL_Const_Iter& src): iter(src.iter),
                end_iter(src.end_iter) {
188             if (*this != src)
```

```
189                 throw std::runtime_error("SDAL_Const_Iter: copy constructor
                        failed");
190             }
191             reference operator*() const {
192                 return *iter;
193             }
194             pointer operator->() const {
195                 return & this->operator*();
196             }
197             self_reference operator=( const self_type& src ) {
198                 if (&src == this)
199                     return *this;
200                 iter = src.iter;
201                 end_iter = src.end_iter;
202                 if (*this != src)
203                     throw std::runtime_error("SDAL_Const_Iter: copy assignment
                            failed");
204                 return *this;
205             }
206             self_reference operator++() { // preincrement
207                 if (iter == end_iter)
208                     throw std::out_of_range("SDAL_Const_Iter: Can't traverse past the
                            end of the list");
209                 ++iter;
210                 return *this;
211             }
212             self_type operator++(int) { // postincrement
213                 self_type t(*this); //save state
214                 operator++(); //apply increment
215                 return t; //return state held before increment
216             }
217             bool operator==(const self_type& rhs) const {
218                 return rhs.iter == iter && rhs.end_iter == end_iter;
219             }
220             bool operator!=(const self_type& rhs) const {
221                 return ! operator==(rhs);
222             }
223         };
224
225         //-------------------------------------------------
226         // types
227         //-------------------------------------------------
228         typedef T value_type;
229         typedef SDAL_Iter iterator;
230         typedef SDAL_Const_Iter const_iterator;
231
232         iterator begin() { return SDAL_Iter(item_array, item_array + num_items); }
233         iterator end() { return SDAL_Iter(item_array + num_items, item_array +
                num_items); }
234
235         const_iterator begin() const { return SDAL_Const_Iter(item_array,
                item_array + num_items); }
```

```
236         const_iterator end() const { return SDAL_Const_Iter(item_array + num_items,
                item_array + num_items); }
237
238         //-------------------------------------------------
239         // operators
240         //-------------------------------------------------
241         T& operator[](size_t i) {
242             if (i >= size()) {
243                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
244             }
245             return item_array[i];
246         }
247
248         const T& operator[](size_t i) const {
249             if (i >= size()) {
250                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
251             }
252             return item_array[i];
253         }
254
255         //-------------------------------------------------
256         // Constructors/destructor/assignment operator
257         //-------------------------------------------------
258
259         SDAL(size_t num_nodes_to_preallocate = 50) {
260             init(num_nodes_to_preallocate);
261         }
262
263         //-------------------------------------------------
264         //copy constructor
265         SDAL(const SDAL& src): SDAL(src.array_size) {
266             init(src.array_size);
267             copy_constructor(src);
268         }
269
270         //-------------------------------------------------
271         //destructor
272         ~SDAL() {
273             // safely dispose of this SDAL's contents
274             delete[] item_array;
275         }
276
277         //-------------------------------------------------
278         //copy assignment constructor
279         SDAL& operator=(const SDAL& src) {
280             if (&src == this) // check for self-assignment
281                 return *this;   // do nothing
282             delete[] item_array;
283             init(src.array_size);
284             copy_constructor(src);
```

```
285             return *this;
286         }
287 
288         //-------------------------------------------------
289         // member functions
290         //-------------------------------------------------
291 
292         /*
293             replaces the existing element at the specified position with the
                    specified element and
294             returns the original element.
295         */
296         T replace(const T& element, size_t position) {
297             T old_item;
298             if (position >= size()) {
299                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
300             } else {
301                 old_item = item_array[position];
302                 item_array[position] = element;
303             }
304             return old_item;
305         }
306 
307         //-------------------------------------------------
308         /*
309             adds the specified element to the list at the specified position,
                    shifting the element
310             originally at that and those in subsequent positions one position to the
                    right.
311         */
312         void insert(const T& element, size_t position) {
313             if (position > size()) {
314                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
315             } else {
316                 embiggen_if_necessary();
317                 //shift remaining items right
318                 for (size_t i = size(); i != position; --i) {
319                     item_array[i] = item_array[i - 1];
320                 }
321                 item_array[position] = element;
322                 ++num_items;
323             }
324         }
325 
326         //-------------------------------------------------
327         //Note to self: use reference here because we receive the original object
                instance,
328         //then copy it into n->item so we have it if the original element goes out
                of scope
329         /*
```

```
330             prepends the specified element to the list.
331         */
332         void push_front(const T& element) {
333             insert(element, 0);
334         }
335 
336         //-------------------------------------------------
337         /*
338             appends the specified element to the list.
339         */
340         void push_back(const T& element) {
341             insert(element, size());
342         }
343 
344 
345         //-------------------------------------------------
346         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
347         //of that so the client still has a valid instance if our destructor is
                called
348         /*
349             removes and returns the element at the list's head.
350         */
351         T pop_front() {
352             if (is_empty()) {
353                 throw std::out_of_range("pop_front: Can't pop: list is empty");
354             }
355             return remove(0);
356         }
357 
358         //-------------------------------------------------
359         /*
360             removes and returns the element at the list's tail.
361         */
362         T pop_back() {
363             if (is_empty()) {
364                 throw std::out_of_range("pop_back: Can't pop: list is empty");
365             }
366             return remove(size() - 1);
367         }
368 
369         //-------------------------------------------------
370         /*
371             removes and returns the the element at the specified position,
372             shifting the subsequent elements one position to the left.
373         */
374         T remove(size_t position) {
375             T item;
376             if (position >= size()) {
377                 throw std::out_of_range(std::string("remove: No element at position
                        ") + std::to_string(position));
378             } else {
```

```
379             item = item_array[position];
380             //shift remaining items left
381             for (size_t i = position + 1; i != size(); ++i) {
382                 item_array[i - 1] = item_array[i];
383             }
384             --num_items;
385             shrink_if_necessary();
386         }
387         return item;
388     }
389 
390     //-------------------------------------------------
391     /*
392         returns (without removing from the list) the element at the specified
                position.
393     */
394     T item_at(size_t position) const {
395         if (position >= size()) {
396             throw std::out_of_range(std::string("item_at: No element at position
                    ") + std::to_string(position));
397         }
398         return operator[](position);
399     }
400 
401     //-------------------------------------------------
402     /*
403         returns true IFF the list contains no elements.
404     */
405     bool is_empty() const {
406         return size() == 0;
407     }
408 
409     //-------------------------------------------------
410     /*
411         returns the number of elements in the list.
412     */
413     size_t size() const {
414         return num_items;
415     }
416 
417     //-------------------------------------------------
418     /*
419         removes all elements from the list.
420     */
421     void clear() {
422         //no reason to do memory deallocation here, just overwrite the old items
                later and save
423         //deallocation for the deconstructor
424         num_items = 0;
425     }
426 
427     //-------------------------------------------------
```

```
428         /*
429             returns true IFF one of the elements of the list matches the specified
                element.
430         */
431         bool contains(const T& element,
432                bool equals(const T& a, const T& b)) const {
433             bool element_in_list = false;
434             const_iterator fin = end();
435             for (const_iterator iter = begin(); iter != fin; ++iter) {
436                 if (equals(*iter, element)) {
437                     element_in_list = true;
438                     break;
439                 }
440             }
441             return element_in_list;
442         }
443 
444         //-------------------------------------------------
445         /*
446             If the list is empty, inserts "<empty list>" into the ostream;
447             otherwise, inserts, enclosed in square brackets, the list's elements,
448             separated by commas, in sequential order.
449         */
450         std::ostream& print(std::ostream& out) const {
451             if (is_empty()) {
452                 out << "<empty list>";
453             } else {
454                 out << "[";
455                 const_iterator start = begin();
456                 const_iterator fin = end();
457                 for (const_iterator iter = start; iter != fin; ++iter) {
458                     if (iter != start)
459                         out << ",";
460                     out << *iter;
461                 }
462                 out << "]";
463             }
464             return out;
465         }
466     protected:
467         bool validate_internal_integrity() {
468             //todo: fill this in
469             return true;
470         }
471     };
472 } // end namespace cop3530
473 
474 #endif // _SDAL_H_
```

CDAL checklist & source code

cdal/checklist.txt

```
Chained Dynamic Array-based List written by Nickerson, Paul
COP 3530, 2014F 1087
=============================================================================
Part I:
=============================================================================
My LIST implementation uses the data structure described in the part I
instructions and conforms to the technique required for this list
variety: yes

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

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes

=============================================================================
Part II:
=============================================================================
My LIST implementation 100% correctly supports the following methods
as described in part II:

* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes

My LIST implementation 100% correctly supports the following data
members as described in part II:

* size_t
* value_type
* iterator
* const_iterator

My ITERATOR implementation 100% correctly supports the following
```

```
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

My CONST ITERATOR implementation 100% correctly supports the following
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My CONST ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

=========================================================================
```

```
Part III:
=========================================================================
My LIST implementation 100% correctly supports the following
methods as described in part III:

* operator[]: yes
* operator[] const: yes

For my LIST's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes
* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes
* operator[]: yes
* operator[] const: yes

For my ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*
```

```
- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

For my CONST ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: yes
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

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

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

My UNIT 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 Chained Dynamic Array-based List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

=====================================================================
In addition to the unit tests, the old_tests directory contains a fuzzer
which stress-tests every list and compares their states to ensure they
all behave equivalently as well as maintain internal integrity.
=====================================================================

How to compile and run my unit tests on the OpenBSD VM
cd list_source_directory
./compile.sh
./unit_tester -s > output.txt
```

cdal/source/CDAL.h

CDAL.h

```
#ifndef _CDAL_H_
#define _CDAL_H_

// CDAL.H
//
// Chained Dynamic Array-based List (non-polymorphic)
//
// Authors: Paul Nickerson, Dave Small
// for COP 3530
// 201409.16 - created

#include <iostream>
#include <stdexcept>
#include <cassert>
#include <math.h>

namespace cop3530 {
    template <class T>
    class CDAL {
    private:
        struct Node {
            //Node is an element in the linked list and contains an array of items
            T* item_array;
            Node* next;
            bool is_dummy;
        };
        struct ItemLoc {
            //ItemLoc describes the position of an item, including its linked list
                node and position within the array held by that node
            Node* node;
            size_t array_index;
            T& item_ref;
        };
        size_t num_items;
        size_t num_available_nodes; //excludes head/tail nodes
        size_t embiggen_counter = 0;
        size_t shrink_counter = 0;
        Node* head;
        Node* tail;
        static const size_t array_size = 50; //length of each chained array
        Node* node_at(size_t position) const {
            Node* n = head->next;
            for (size_t i = 0; i != position; ++i, n = n->next);
            return n;
        }
        Node* node_before(size_t position) const {
            if (position == 0)
```

```
            return head;
        else
            return node_at(position - 1);
    }

    ItemLoc loc_from_pos(size_t position) const {
        size_t node_position = floor(position / array_size);
        Node* n = node_at(node_position);
        size_t array_index = position % array_size;
        ItemLoc loc {n, array_index, n->item_array[array_index]};
        return loc;
    }

    Node* design_new_node(Node* next = nullptr, bool dummy = false) const {
        Node* n = new Node();
        n->is_dummy = dummy;
        n->item_array = new T[array_size];
        n->next = next;
        return n;
    }

    void init() {
        num_items = 0;
        num_available_nodes = 0;
        tail = design_new_node(nullptr, true);
        head = design_new_node(tail, true);
    }

    void free_node(Node* n) {
        delete[] n->item_array;
        delete n;
    }

    void drop_node_after(Node* n) {
        assert(n->next != tail);
        Node* removed_node = n->next;
        n->next = removed_node->next;
        free_node(removed_node);
        --num_available_nodes;
    }

    size_t num_used_nodes() {
        return ceil(size() / array_size);
    }

    void embiggen_if_necessary() {
        //embiggen is a perfectly cromulent word
        /*
            If each array slot in every link is filled and we want to add a new
                item, allocate and append a new link
        */
        if (size() == num_available_nodes * array_size) {
```

```
98              //transform tail into a regular node and append a new tail
99              Node* n = tail;
100             n->is_dummy = false;
101             tail = n->next = design_new_node(nullptr, false);
102             ++num_available_nodes;
103             ++embiggen_counter;
104         }
105     }
106
107     void shrink_if_necessary() {
108         /*
109             Because we don't want the list to waste too much memory, whenever
                the more than half of the arrays
110             are unused (they would all be at the end of the chain), deallocate
                half the unused arrays.
111         */
112         size_t used = num_used_nodes();
113         size_t num_unused_nodes = num_available_nodes - used;
114         if (num_unused_nodes > used) {
115             size_t nodes_to_keep = used + ceil(num_unused_nodes * 0.5);
116             Node* last_node = node_before(nodes_to_keep);
117             while (last_node->next != tail) {
118                 drop_node_after(last_node);
119             }
120             ++shrink_counter;
121         }
122     }
123     void copy_constructor(const CDAL& src) {
124         const_iterator fin = src.end();
125         for (const_iterator iter = src.begin(); iter != fin; ++iter) {
126             push_back(*iter);
127         }
128         if ( ! src.size() == size())
129             throw std::runtime_error("copy_constructor: Copying failed - sizes
                    don't match up");
130     }
131
132 public:
133     //-------------------------------------------------
134     // iterators
135     //-------------------------------------------------
136     class CDAL_Iter: public std::iterator<std::forward_iterator_tag, T> {
137     private:
138         Node* here_container;
139         size_t here_index;
140     public:
141         typedef std::ptrdiff_t difference_type;
142         typedef T& reference;
143         typedef T* pointer;
144         typedef std::forward_iterator_tag iterator_category;
145         typedef T value_type;
146         typedef CDAL_Iter self_type;
```

```
147             typedef CDAL_Iter& self_reference;
148
149             //need copy constructor/assigner to make this a first class ADT (doesn't
                    hold pointers that need freeing)
150             CDAL_Iter(Node* container, size_t index): here_container(container),
                    here_index(index) {}
151             CDAL_Iter(const self_type& src): here_container(src.here_container),
                    here_index(src.here_index) {}
152             self_reference operator=(const self_type& rhs) {
153                 //copy assigner
154                 if (&rhs == this) return *this;
155                 here_container = rhs.here_container;
156                 here_index = rhs.here_index;
157                 return this;
158             }
159             self_reference operator++() {
160                 //prefix (no int parameter)
161                 here_index = (here_index + 1) % array_size;
162                 if (here_index == 0) here_container = here_container->next;
163                 return *this;
164             }
165             self_type operator++(int) { // postincrement
166                 self_type t(*this); //save state
167                 operator++(); //apply increment
168                 return t; //return state held before increment
169             }
170             reference operator*() const {
171                 return here_container->item_array[here_index];
172             }
173             pointer operator->() const {
174                 return & this->operator*();
175             }
176             bool operator==(const self_type& rhs) const {
177                 return rhs.here_index == here_index
178                         && rhs.here_container == here_container;
179             }
180             bool operator!=(const self_type& rhs) const {
181                 return ! operator==(rhs);
182             }
183         };
184
185         class CDAL_Const_Iter: public std::iterator<std::forward_iterator_tag, T> {
186         private:
187             const Node* here_container;
188             size_t here_index;
189         public:
190             //todo: check on whether value_type should/shouldn't be const
191             typedef const T value_type;
192             typedef const T& reference;
193             typedef const T* pointer;
194             typedef std::forward_iterator_tag iterator_category;
195             typedef std::ptrdiff_t difference_type;
```

```
196             typedef CDAL_Const_Iter self_type;
197             typedef CDAL_Const_Iter& self_reference;
198 
199             CDAL_Const_Iter(const Node* container, size_t index):
                    here_container(container), here_index(index) {}
200             CDAL_Const_Iter(const self_type& src):
                    here_container(src.here_container), here_index(src.here_index) {}
201             self_reference operator=(const self_type& rhs) {
202                 //copy assigner
203                 if (&rhs == this) return *this;
204                 here_container = rhs.here_container;
205                 here_index = rhs.here_index;
206                 return this;
207             }
208             self_reference operator++() {
209                 //prefix (no int parameter)
210                 here_index = (here_index + 1) % array_size;
211                 if (here_index == 0) here_container = here_container->next;
212                 return *this;
213             }
214             self_type operator++(int) { // postincrement
215                 self_type t(*this); //save state
216                 operator++(); //apply increment
217                 return t; //return state held before increment
218             }
219             reference operator*() const {
220                 return here_container->item_array[here_index];
221             }
222             pointer operator->() const {
223                 return & this->operator*();
224             }
225             bool operator==(const self_type& rhs) const {
226                 return rhs.here_index == here_index
227                        && rhs.here_container == here_container;
228             }
229             bool operator!=(const self_type& rhs) const {
230                 return ! operator==(rhs);
231             }
232         };
233 
234         //-------------------------------------------------
235         // types
236         //-------------------------------------------------
237         typedef CDAL_Iter iterator;
238         typedef CDAL_Const_Iter const_iterator;
239         typedef T value_type;
240         //todo: might need to add size_t here and other iterators if they were
                excluded or commented out
241 
242         iterator begin() {
243             return iterator(head->next, 0);
244         }
```

```
245
246         iterator end() {
247             ItemLoc end_loc = loc_from_pos(size());
248             return iterator(end_loc.node, end_loc.array_index);
249         }
250
251         const_iterator begin() const {
252             return const_iterator(head->next, 0);
253         }
254
255         const_iterator end() const {
256             ItemLoc end_loc = loc_from_pos(size());
257             return const_iterator(end_loc.node, end_loc.array_index);
258         }
259
260         T& operator[](size_t i) {
261             if (i >= size()) {
262                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
263             }
264             return loc_from_pos(i).item_ref;
265         }
266
267         const T& operator[](size_t i) const {
268             if (i >= size()) {
269                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
270             }
271             return loc_from_pos(i).item_ref;
272         }
273
274         //-------------------------------------------------
275         // Constructors/destructor/assignment operator
276         //-------------------------------------------------
277
278         CDAL() {
279             init();
280             embiggen_if_necessary();
281         }
282         //-------------------------------------------------
283         //copy constructor
284         CDAL(const CDAL& src) {
285             init();
286             copy_constructor(src);
287         }
288
289         //-------------------------------------------------
290         //destructor
291         ~CDAL() {
292             // safely dispose of this CDAL's contents
293             clear();
294         }
```

```
295
296         //-------------------------------------------------
297         //copy assignment constructor
298         CDAL& operator=(const CDAL& src) {
299             if (&src == this) // check for self-assignment
300                 return *this;   // do nothing
301             // safely dispose of this CDAL's contents
302             // populate this CDAL with copies of the other CDAL's contents
303             clear();
304             init();
305             copy_constructor(src);
306             return *this;
307         }
308
309         //-------------------------------------------------
310         // member functions
311         //-------------------------------------------------
312
313         /*
314             replaces the existing element at the specified position with the
                    specified element and
315             returns the original element.
316         */
317         T replace(const T& element, size_t position) {
318             T item = element;
319             if (position >= size()) {
320                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
321             } else {
322                 ItemLoc loc = loc_from_pos(position);
323                 std::swap(loc.item_ref, item);
324             }
325             return item;
326         }
327
328         //-------------------------------------------------
329         /*
330             adds the specified element to the list at the specified position,
                    shifting the element
331             originally at that and those in subsequent positions one position to the
                    right.
332         */
333         void insert(const T& element, size_t position) {
334             if (position > size()) {
335                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
336             } else {
337                 embiggen_if_necessary();
338                 ItemLoc loc = loc_from_pos(position);
339                 //shift remaining items to the right
340                 T item_to_insert = element;
341                 Node* n = loc.node;
```

```
342                 for (size_t i = position; i <= num_items; ++i) {
343                     size_t array_index = i % array_size;
344                     if ( i != position && array_index == 0 ) {
345                         n = n->next;
346                     }
347                     std::swap(item_to_insert, n->item_array[array_index]);
348                 }
349                 ++num_items;
350             }
351         }
352 
353         //-------------------------------------------------
354         //Note to self: use reference here because we receive the original object
                instance,
355         //then copy it into n->item so we have it if the original element goes out
                of scope
356         /*
357             prepends the specified element to the list.
358         */
359         void push_front(const T& element) {
360             insert(element, 0);
361         }
362 
363         //-------------------------------------------------
364         /*
365             appends the specified element to the list.
366         */
367         void push_back(const T& element) {
368             insert(element, size());
369         }
370 
371         //-------------------------------------------------
372         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
373         //of that so the client still has a valid instance if our destructor is
                called
374         /*
375             removes and returns the element at the list's head.
376         */
377         T pop_front() {
378             if (is_empty()) {
379                 throw std::out_of_range("pop_front: Can't pop: list is empty");
380             }
381             return remove(0);
382         }
383 
384         //-------------------------------------------------
385         /*
386             removes and returns the element at the list's tail.
387         */
388         T pop_back() {
389             if (is_empty()) {
```

```
390             throw std::out_of_range("pop_back: Can't pop: list is empty");
391         }
392         return remove(size() - 1);
393     }
394 
395     //-------------------------------------------------
396     /*
397         removes and returns the the element at the specified position,
398         shifting the subsequent elements one position to the left.
399     */
400     T remove(size_t position) {
401         T old_item;
402         if (position >= size()) {
403             throw std::out_of_range(std::string("remove: No element at position
                    ") + std::to_string(position));
404         } else {
405             ItemLoc loc = loc_from_pos(position);
406             //shift remaining items to the left
407             Node* n = loc.node;
408             old_item = loc.item_ref;
409             for (size_t i = position; i != num_items; ++i) {
410                 size_t curr_array_index = i % array_size;
411                 size_t next_array_index = (i + 1) % array_size;
412                 T& curr_item = n->item_array[curr_array_index];
413                 if ( next_array_index == 0 ) {
414                     n = n->next;
415                 }
416                 T& next_item = n->item_array[next_array_index];
417                 std::swap(curr_item, next_item);
418             }
419             --num_items;
420             shrink_if_necessary();
421         }
422         return old_item;
423     }
424 
425     //-------------------------------------------------
426     /*
427         returns (without removing from the list) the element at the specified
                position.
428     */
429     T item_at(size_t position) const {
430         if (position >= size()) {
431             throw std::out_of_range(std::string("item_at: No element at position
                    ") + std::to_string(position));
432         }
433         return loc_from_pos(position).item_ref;
434     }
435 
436     //-------------------------------------------------
437     /*
438         returns true IFF the list contains no elements.
```

```
439         */
440         bool is_empty() const {
441             return size() == 0;
442         }
443 
444         //-------------------------------------------------
445         /*
446             returns the number of elements in the list.
447         */
448         size_t size() const {
449             return num_items;
450         }
451 
452         //-------------------------------------------------
453         /*
454             removes all elements from the list.
455         */
456         void clear() {
457             while (head->next != tail) {
458                 drop_node_after(head);
459             }
460             num_items = 0;
461         }
462         //-------------------------------------------------
463         /*
464             returns true IFF one of the elements of the list matches the specified
                 element.
465         */
466         bool contains(const T& element,
467                bool equals(const T& a, const T& b)) const {
468             bool element_in_list = false;
469             const_iterator fin = end();
470             for (const_iterator iter = begin(); iter != fin; ++iter) {
471                 if (equals(*iter, element)) {
472                     element_in_list = true;
473                     break;
474                 }
475             }
476             return element_in_list;
477         }
478 
479         //-------------------------------------------------
480         /*
481             If the list is empty, inserts "<empty list>" into the ostream;
482             otherwise, inserts, enclosed in square brackets, the list's elements,
483             separated by commas, in sequential order.
484         */
485         std::ostream& print(std::ostream& out) const {
486             if (is_empty()) {
487                 out << "<empty list>";
488             } else {
489                 out << "[";
```

```
490                 const_iterator start = begin();
491                 const_iterator fin = end();
492                 for (const_iterator iter = start; iter != fin; ++iter) {
493                     if (iter != start)
494                         out << ",";
495                     out << *iter;
496                 }
497                 out << "]";
498             }
499             return out;
500         }
501     protected:
502         bool validate_internal_integrity() {
503             //todo: fill this in
504             return true;
505         }
506     }; //end class CDAL
507 } // end namespace cop3530
508 #endif // _CDAL_H_
```