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 current 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 current 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 a non-const iterator (so we can use references to avoid copy constructors) 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

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the current 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 current 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 a non-const iterator (so we can use references to avoid copy constructors) 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

SDAL

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the current 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 current 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 a non-const iterator (so we can use references to avoid copy constructors) 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

CDAL

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the current 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 current 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 a non-const iterator (so we can use references to avoid copy constructors) 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      }
95  public:
```

```
96
97          //-------------------------------------------------
98          // iterators
99          //-------------------------------------------------
100         class SSLL_Iter: public std::iterator<std::forward_iterator_tag, T>
101         {
102         public:
103             // inheriting from std::iterator<std::forward_iterator_tag, T>
104             // automagically sets up these typedefs...
105             typedef T value_type;
106             typedef std::ptrdiff_t difference_type;
107             typedef T& reference;
108             typedef T* pointer;
109             typedef std::forward_iterator_tag iterator_category;
110
111             // but not these typedefs...
112             typedef SSLL_Iter self_type;
113             typedef SSLL_Iter& self_reference;
114
115         private:
116             Node* here;
117
118         public:
119             explicit SSLL_Iter(Node* start) : here(start) {
120                 if (start == nullptr)
121                     throw std::runtime_error("SSLL_Iter: start cannot be null");
122             }
123             SSLL_Iter(const SSLL_Iter& src) : here(src.here) {
124                 if (*this != src)
125                     throw std::runtime_error("SSLL_Iter: copy constructor failed");
126             }
127             reference operator*() const {
128                 return here->item;
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                 here = src.here;
137                 if (*this != src)
138                     throw std::runtime_error("SSLL_Iter: copy assignment failed");
139                 return *this;
140             }
141             self_reference operator++() { // preincrement
142                 if (here->next == nullptr)
143                     throw std::out_of_range("SSLL_Iter: Can't traverse past the end
                        of the list");
144                 here = here->next;
145                 return *this;
146             }
```

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

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

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

```
295             if (position >= size()) {
296                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
297             } else {
298                 //we are guaranteed to be at a non-dummy item now because of the
                        above if statement
299                 Node* iter = node_at(position);
300                 old_item = iter->item;
301                 iter->item = element;
302             }
303             return old_item;
304         }
305 
306         //-------------------------------------------------
307         /*
308             adds the specified element to the list at the specified position,
                    shifting the element
309             originally at that and those in subsequent positions one position to the
                    right.
310         */
311         void insert(const T& element, size_t position) {
312             if (position > size()) {
313                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
314             } else if (position == size()) {
315                 //special O(1) case
316                 push_back(element);
317             } else {
318                 //node_before_position is guaranteed to point to a valid node
                        because we use a dummy head node
319                 Node* node_before_position = node_before(position);
320                 Node* node_at_position = node_before_position->next;
321                 Node* new_node;
322                 try {
323                     new_node = design_new_node(element, node_at_position);
324                 } catch (std::bad_alloc& ba) {
325                     std::cerr << "insert(): failed to allocate memory for new node"
                            << std::endl;
326                     throw std::bad_alloc();
327                 }
328                 insert_node_after(node_before_position, new_node);
329             }
330         }
331 
332         /*
333             prepends the specified element to the list.
334         */
335         void push_front(const T& element) {
336             insert(element, 0);
337         }
338 
339         //-------------------------------------------------
```

```
340         /*
341             appends the specified element to the list.
342         */
343         void push_back(const T& element) {
344             Node* new_tail;
345             try {
346                 new_tail = design_new_node(nullptr, true);
347             } catch (std::bad_alloc& ba) {
348                 std::cerr << "push_back(): failed to allocate memory for new tail"
                        << std::endl;
349                 throw std::bad_alloc();
350             }
351             insert_node_after(tail, new_tail);
352             //transform the current tail node from a dummy to a real node holding
                    element
353             tail->is_dummy = false;
354             tail->item = element;
355             tail->next = new_tail;
356             tail = tail->next;
357         }
358
359         /*
360             removes and returns the element at the list's head.
361         */
362         T pop_front() {
363             if (is_empty()) {
364                 throw std::out_of_range("pop_front: Can't pop: list is empty");
365             }
366             if (head->next == tail) {
367                 throw std::runtime_error("pop_front: head->next == tail, but list
                        says it's not empty (corrupt state)");
368             }
369             return remove_item_after(head);
370         }
371
372         //-------------------------------------------------
373         /*
374             removes and returns the element at the list's tail.
375         */
376         T pop_back() {
377             if (is_empty()) {
378                 throw std::out_of_range("pop_back: Can't pop: list is empty");
379             }
380             if (head->next == tail) {
381                 throw std::runtime_error("pop_back: head->next == tail, but list
                        says it's not empty (corrupt state)");
382             }
383             //XXX this is O(N), a disadvantage of this architecture
384             Node* node_before_last = node_before(size() - 1);
385             T item = remove_item_after(node_before_last);
386             return item;
387         }
```

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

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

```
484                 }
485                 out << "]";
486             }
487             return out;
488         }
489     protected:
490         bool validate_internal_integrity() {
491             //todo: fill this in
492             return true;
493         }
494     }; //end class SSLL
495 } // end namespace cop3530
496 #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) {
                n = new Node();
            } else {
                n = remove_node_after(free_list_head, num_free_list_items);
```

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

```
97          }
98
99          void insert_node_after(Node* existing_node, Node* new_node, size_t&
                list_size_counter) {
100             new_node->next = existing_node->next;
101             existing_node->next = new_node;
102             ++list_size_counter;
103         }
104
105         //returns subsequent node's item and moves that node to the free pool
106         T remove_item_after(Node* preceeding_node) {
107             Node* removed_node = remove_node_after(preceeding_node,
                    num_main_list_items);
108             T item = removed_node->item;
109             insert_node_after(free_list_head, removed_node, num_free_list_items);
110             shrink_pool_if_necessary();
111             return item;
112         }
113
114     public:
115         //-------------------------------------------------
116         // iterators
117         //-------------------------------------------------
118         class PSLL_Iter: public std::iterator<std::forward_iterator_tag, T> {
119         private:
120             Node* here;
121         public:
122             typedef T value_type;
123             typedef std::ptrdiff_t difference_type;
124             typedef T* pointer;
125             typedef T& reference;
126             typedef std::forward_iterator_tag iterator_category;
127
128             typedef PSLL_Iter self_type;
129             typedef PSLL_Iter& self_reference;
130
131             explicit PSLL_Iter(Node* start): here(start) {
132                 if (start == nullptr)
133                     throw std::runtime_error("PSLL_Iter: start cannot be null");
134             }
135             PSLL_Iter(const self_type& src): here(src.here) {}
136
137             reference operator*() const {
138                 return here->item;
139             }
140             pointer operator->() const {
141                 return & this->operator*();
142             }
143             self_reference operator=(const self_type& src) {
144                 //copy assigner
145                 if (&src == this)
146                     return *this;
```

```
147             here = src.here;
148             return *this;
149         }
150         self_reference operator++() {
151             //prefix
152             here = here->next;
153             return *this;
154         }
155         self_type operator++(int) {
156             self_type t(*this); //save state
157             operator++(); //apply increment
158             return t; //return state held before increment
159         }
160         bool operator==(const self_type& rhs) const {
161             return here == rhs.here;
162         }
163         bool operator!=(const self_type& rhs) const {
164             return ! operator==(rhs);
165         }
166     };
167
168     class PSLL_Const_Iter: public std::iterator<std::forward_iterator_tag, T> {
169     private:
170         const Node* here;
171     public:
172         typedef T value_type;
173         typedef std::ptrdiff_t difference_type;
174         typedef const T* pointer;
175         typedef const T& reference;
176         typedef std::forward_iterator_tag iterator_category;
177
178         typedef PSLL_Const_Iter self_type;
179         typedef PSLL_Const_Iter& self_reference;
180
181         explicit PSLL_Const_Iter(Node* start): here(start) {
182             if (start == nullptr)
183                 throw std::runtime_error("PSLL_Const_Iter: start cannot be null");
184         }
185         PSLL_Const_Iter(const self_type& src): here(src.here) {}
186
187         reference operator*() const {
188             return here->item;
189         }
190         pointer operator->() const {
191             return & this->operator*();
192         }
193         self_reference operator=(const self_type& src) {
194             //copy assigner
195             if (&src == this)
196                 return *this;
197             here = src.here;
198             return *this;
```

```
199             }
200             self_reference operator++() {
201                 //prefix
202                 here = here->next;
203                 return *this;
204             }
205             self_type operator++(int) {
206                 self_type t(*this); //save state
207                 operator++(); //apply increment
208                 return t; //return state held before increment
209             }
210             bool operator==(const self_type& rhs) const {
211                 return here == rhs.here;
212             }
213             bool operator!=(const self_type& rhs) const {
214                 return ! operator==(rhs);
215             }
216         };
217 
218         //-------------------------------------------------
219         // types
220         //-------------------------------------------------
221         /*typedef std::size_t size_t;*/
222         typedef T value_type;
223         typedef PSLL_Iter iterator;
224         typedef PSLL_Const_Iter const_iterator;
225 
226         iterator begin() {
227             return iterator(head->next);
228         }
229         iterator end() {
230             return iterator(tail);
231         }
232         /*
233             Note to self: the following overloads will fail if not defined as const
234         */
235         const_iterator begin() const {
236             return const_iterator(head->next);
237         }
238         const_iterator end() const {
239             return const_iterator(tail);
240         }
241 
242         //-------------------------------------------------
243         // operators
244         //-------------------------------------------------
245         T& operator[](size_t i) {
246             if (i >= size()) {
247                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
248             }
249             return node_at(i)->item;
```

```
250         }
251
252         const T& operator[](size_t i) const {
253             if (i >= size()) {
254                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
255             }
256             return node_at(i)->item;
257         }
258
259         //-------------------------------------------------
260         // Constructors/destructor/assignment operator
261         //-------------------------------------------------
262
263         PSLL() {
264             init();
265         }
266         //-------------------------------------------------
267         //copy constructor
268         PSLL(const PSLL& src) {
269             init();
270             copy_constructor(src);
271         }
272
273         //-------------------------------------------------
274         //destructor
275         ~PSLL() {
276             // safely dispose of this PSLL's contents
277             clear();
278         }
279
280         //-------------------------------------------------
281         //copy assignment constructor
282         PSLL& operator=(const PSLL& src) {
283             if (&src == this) // check for self-assignment
284                 return *this;   // do nothing
285             // safely dispose of this PSLL's contents
286             // populate this PSLL with copies of the other PSLL's contents
287             clear();
288             copy_constructor(src);
289             return *this;
290         }
291
292         //-------------------------------------------------
293         // member functions
294         //-------------------------------------------------
295
296         /*
297             replaces the existing element at the specified position with the
                    specified element and
298             returns the original element.
299         */
```

```
300         T replace(const T& element, size_t position) {
301             T old_item;
302             if (position >= size()) {
303                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
304             } else {
305                 //we are guaranteed to be at a non-dummy item now because of the
                        above if statement
306                 Node* iter = node_at(position);
307                 old_item = iter->item;
308                 iter->item = element;
309             }
310             return old_item;
311         }
312 
313         //-------------------------------------------------
314         /*
315             adds the specified element to the list at the specified position,
                    shifting the element
316             originally at that and those in subsequent positions one position to the
                    right.
317         */
318         void insert(const T& element, size_t position) {
319             if (position > size()) {
320                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
321             } else {
322                 //node_before_position is guaranteed to point to a valid node
                        because we use a dummy head node
323                 Node* node_before_position = node_before(position);
324                 Node* node_at_position = node_before_position->next;
325                 Node* new_node = design_new_node(element, node_at_position);
326                 insert_node_after(node_before_position, new_node,
                        num_main_list_items);
327             }
328         }
329 
330         //-------------------------------------------------
331         //Note to self: use reference here because we receive the original object
                instance,
332         //then copy it into n->item so we have it if the original element goes out
                of scope
333         /*
334             prepends the specified element to the list.
335         */
336         void push_front(const T& element) {
337             insert(element, 0);
338         }
339 
340         //-------------------------------------------------
341         /*
342             appends the specified element to the list.
```

```
343         */
344         void push_back(const T& element) {
345             Node* new_tail = design_new_node(nullptr, true);
346             insert_node_after(tail, new_tail, num_main_list_items);
347             //transform the current tail node from a dummy to a real node holding
                    element
348             tail->is_dummy = false;
349             tail->item = element;
350             tail->next = new_tail;
351             tail = tail->next;
352         }
353
354         //-------------------------------------------------
355         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
356         //of that so the client still has a valid instance if our destructor is
                called
357         /*
358             removes and returns the element at the list's head.
359         */
360         T pop_front() {
361             if (is_empty()) {
362                 throw std::out_of_range("pop_front: Can't pop: list is empty");
363             }
364             return remove_item_after(head);
365         }
366
367         //-------------------------------------------------
368         /*
369             removes and returns the element at the list's tail.
370         */
371         T pop_back() {
372             if (is_empty()) {
373                 throw std::out_of_range("pop_back: Can't pop: list is empty");
374             }
375             //XXX this is O(N), a disadvantage of this architecture
376             Node* node_before_last = node_before(size() - 1);
377             T item = remove_item_after(node_before_last);
378             return item;
379         }
380
381         //-------------------------------------------------
382         /*
383             removes and returns the the element at the specified position,
384             shifting the subsequent elements one position to the left.
385         */
386         T remove(size_t position) {
387             T item;
388             if (position >= size()) {
389                 throw std::out_of_range(std::string("remove: No element at position
                        ") + std::to_string(position));
390             } else {
```

```
391             //using a dummy head node guarantees that there be a node
                    immediately preceeding the specified position
392             Node *node_before_position = node_before(position);
393             item = remove_item_after(node_before_position);
394         }
395         return item;
396     }
397 
398     //-------------------------------------------------
399     /*
400         returns (without removing from the list) the element at the specified
                position.
401     */
402     T item_at(size_t position) const {
403         if (position >= size()) {
404             throw std::out_of_range(std::string("item_at: No element at position
                    ") + std::to_string(position));
405         }
406         return node_at(position)->item;
407     }
408 
409     //-------------------------------------------------
410     /*
411         returns true IFF the list contains no elements.
412     */
413     bool is_empty() const {
414         return size() == 0;
415     }
416 
417     //-------------------------------------------------
418     /*
419         returns the number of elements in the list.
420     */
421     size_t size() const {
422         assert( ! (num_main_list_items == 0 && head->next != tail));
423         return num_main_list_items;
424     }
425 
426     //-------------------------------------------------
427     /*
428         removes all elements from the list.
429     */
430     void clear() {
431         while (size()) {
432             remove_item_after(head);
433         }
434     }
435     //-------------------------------------------------
436     /*
437         returns true IFF one of the elements of the list matches the specified
                element.
438     */
```

```
439         bool contains(const T& element,
440               bool equals(const T& a, const T& b)) const {
441             bool element_in_list = false;
442             const_iterator fin = end();
443             for (const_iterator iter = begin(); iter != fin; ++iter) {
444                 if (equals(*iter, element)) {
445                     element_in_list = true;
446                     break;
447                 }
448             }
449             return element_in_list;
450         }
451 
452         //-------------------------------------------------
453         /*
454             If the list is empty, inserts "<empty list>" into the ostream;
455             otherwise, inserts, enclosed in square brackets, the list's elements,
456             separated by commas, in sequential order.
457         */
458         std::ostream& print(std::ostream& out) const {
459             if (is_empty()) {
460                 out << "<empty list>";
461             } else {
462                 out << "[";
463                 const_iterator start = begin();
464                 const_iterator fin = end();
465                 for (const_iterator iter = start; iter != fin; ++iter) {
466                     if (iter != start)
467                         out << ",";
468                     out << *iter;
469                 }
470                 out << "]";
471             }
472             return out;
473         }
474     protected:
475         bool validate_internal_integrity() {
476             //todo: fill this in
477             return true;
478         }
479     }; //end class PSLL
480 } // end namespace cop3530
481 #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;
        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) {
                size_t new_array_size = ceil(array_size * 1.5);
                T* new_item_array = new T[new_array_size];
                for (size_t i = 0; i != filled_slots; ++i) {
                    new_item_array[i] = item_array[i];
                }
                delete[] item_array;
                item_array = new_item_array;
                array_size = new_array_size;
                ++embiggen_counter;
            }
```

```
46          }
47          void shrink_if_necessary() {
48              /*
49                  Because we don't want the list to waste too much memory, whenever
                        the array's size is  100 slots
50                  and fewer than half the slots are used, allocate a new array 50% the
                        size of the original, copy
51                  the items over to the new array, and deallocate the original one.
52              */
53              size_t filled_slots = size();
54              if (array_size >= 100 && filled_slots < array_size / 2) {
55                  size_t new_array_size = ceil(array_size * 0.5);
56                  T* new_item_array = new T[new_array_size];
57                  for (size_t i = 0; i != filled_slots; ++i) {
58                      new_item_array[i] = item_array[i];
59                  }
60                  delete[] item_array;
61                  item_array = new_item_array;
62                  array_size = new_array_size;
63                  ++shrink_counter;
64              }
65          }
66          void init(size_t num_nodes_to_preallocate) {
67              array_size = num_nodes_to_preallocate;
68              num_items = 0;
69              item_array = new T[array_size];
70          }
71          void copy_constructor(const SDAL& src) {
72              const_iterator fin = src.end();
73              for (const_iterator iter = src.begin(); iter != fin; ++iter) {
74                  push_back(*iter);
75              }
76          }
77      public:
78
79          //------------------------------------------------
80          // iterators
81          //------------------------------------------------
82          class SDAL_Iter: public std::iterator<std::forward_iterator_tag, T>
83          {
84          public:
85              // inheriting from std::iterator<std::forward_iterator_tag, T>
86              // automagically sets up these typedefs...
87              //todo: figure out why we cant comment these out, which we should be
                        able to if they were
88              //defined when inheriting
89              typedef T value_type;
90              typedef std::ptrdiff_t difference_type;
91              typedef T& reference;
92              typedef T* pointer;
93              typedef std::forward_iterator_tag iterator_category;
94
```

```
95              // but not these typedefs...
96              typedef SDAL_Iter self_type;
97              typedef SDAL_Iter& self_reference;
98
99          private:
100             T* iter;
101
102         public:
103             explicit SDAL_Iter(T* item_array) : iter(item_array) {}
104             SDAL_Iter(const SDAL_Iter& src) : iter(src.iter) {}
105
106             reference operator*() const {
107                 return *iter;
108             }
109             pointer operator->() const {
110                 return & this->operator*();
111             }
112             self_reference operator=( const self_type& src ) {
113                 if (&src == this)
114                     return *this;
115                 iter = src.iter;
116                 return *this;
117             }
118             self_reference operator++() { // preincrement
119                 ++iter;
120                 return *this;
121             }
122             self_type operator++(int) { // postincrement
123                 self_type t(*this); //save state
124                 operator++(); //apply increment
125                 return t; //return state held before increment
126             }
127             bool operator==(const self_type& rhs) const {
128                 return rhs.iter == iter;
129             }
130             bool operator!=(const self_type& rhs) const {
131                 return ! operator==(rhs);
132             }
133         };
134
135         class SDAL_Const_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 const T& reference;
143             typedef const T* pointer;
144             typedef std::forward_iterator_tag iterator_category;
145
146             // but not these typedefs...
```

```
147             typedef SDAL_Const_Iter self_type;
148             typedef SDAL_Const_Iter& self_reference;
149         private:
150             const T* iter;
151         public:
152             explicit SDAL_Const_Iter(T* item_array) : iter(item_array) {}
153             SDAL_Const_Iter(const SDAL_Const_Iter& src) : iter(src.iter) {}
154 
155             reference operator*() const {
156                 return *iter;
157             }
158             pointer operator->() const {
159                 return & this->operator*();
160             }
161             self_reference operator=( const self_type& src ) {
162                 if (&src == this)
163                     return *this;
164                 iter = src.iter;
165                 return *this;
166             }
167             self_reference operator++() { // preincrement
168                 ++iter;
169                 return *this;
170             }
171             self_type operator++(int) { // postincrement
172                 self_type t(*this); //save state
173                 operator++(); //apply increment
174                 return t; //return state held before increment
175             }
176             bool operator==(const self_type& rhs) const {
177                 return rhs.iter == iter;
178             }
179             bool operator!=(const self_type& rhs) const {
180                 return ! operator==(rhs);
181             }
182         };
183 
184         //-------------------------------------------------
185         // types
186         //-------------------------------------------------
187         typedef T value_type;
188         typedef SDAL_Iter iterator;
189         typedef SDAL_Const_Iter const_iterator;
190 
191         iterator begin() { return SDAL_Iter(item_array); }
192         iterator end() { return SDAL_Iter(item_array + num_items); }
193 
194         const_iterator begin() const { return SDAL_Const_Iter(item_array); }
195         const_iterator end() const { return SDAL_Const_Iter(item_array +
                num_items); }
196 
197         //-------------------------------------------------
```

```
198         // operators
199         //-------------------------------------------------
200         T& operator[](size_t i) {
201             if (i >= size()) {
202                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
203             }
204             return item_array[i];
205         }
206
207         const T& operator[](size_t i) const {
208             if (i >= size()) {
209                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
210             }
211             return item_array[i];
212         }
213
214         //-------------------------------------------------
215         // Constructors/destructor/assignment operator
216         //-------------------------------------------------
217
218         SDAL(size_t num_nodes_to_preallocate = 50) {
219             init(num_nodes_to_preallocate);
220         }
221
222         //-------------------------------------------------
223         //copy constructor
224         SDAL(const SDAL& src): SDAL(src.array_size) {
225             init(src.array_size);
226             copy_constructor(src);
227         }
228
229         //-------------------------------------------------
230         //destructor
231         ~SDAL() {
232             // safely dispose of this SDAL's contents
233             delete[] item_array;
234         }
235
236         //-------------------------------------------------
237         //copy assignment constructor
238         SDAL& operator=(const SDAL& src) {
239             if (&src == this) // check for self-assignment
240                 return *this;   // do nothing
241             delete[] item_array;
242             init(src.array_size);
243             copy_constructor(src);
244             return *this;
245         }
246
247         //-------------------------------------------------
```

```
248         // member functions
249         //-------------------------------------------------
250
251         /*
252             replaces the existing element at the specified position with the
                specified element and
253             returns the original element.
254         */
255         T replace(const T& element, size_t position) {
256             T old_item;
257             if (position >= size()) {
258                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
259             } else {
260                 old_item = item_array[position];
261                 item_array[position] = element;
262             }
263             return old_item;
264         }
265
266         //-------------------------------------------------
267         /*
268             adds the specified element to the list at the specified position,
                shifting the element
269             originally at that and those in subsequent positions one position to the
                right.
270         */
271         void insert(const T& element, size_t position) {
272             if (position > size()) {
273                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
274             } else {
275                 embiggen_if_necessary();
276                 //shift remaining items right
277                 for (size_t i = size(); i != position; --i) {
278                     item_array[i] = item_array[i - 1];
279                 }
280                 item_array[position] = element;
281                 ++num_items;
282             }
283         }
284
285         //-------------------------------------------------
286         //Note to self: use reference here because we receive the original object
                instance,
287         //then copy it into n->item so we have it if the original element goes out
                of scope
288         /*
289             prepends the specified element to the list.
290         */
291         void push_front(const T& element) {
292             insert(element, 0);
```

```
293         }
294
295         //-------------------------------------------------
296         /*
297             appends the specified element to the list.
298         */
299         void push_back(const T& element) {
300             insert(element, size());
301         }
302
303
304         //-------------------------------------------------
305         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
306         //of that so the client still has a valid instance if our destructor is
                called
307         /*
308             removes and returns the element at the list's head.
309         */
310         T pop_front() {
311             if (is_empty()) {
312                 throw std::out_of_range("pop_front: Can't pop: list is empty");
313             }
314             return remove(0);
315         }
316
317         //-------------------------------------------------
318         /*
319             removes and returns the element at the list's tail.
320         */
321         T pop_back() {
322             if (is_empty()) {
323                 throw std::out_of_range("pop_back: Can't pop: list is empty");
324             }
325             return remove(size() - 1);
326         }
327
328         //-------------------------------------------------
329         /*
330             removes and returns the the element at the specified position,
331             shifting the subsequent elements one position to the left.
332         */
333         T remove(size_t position) {
334             T item;
335             if (position >= size()) {
336                 throw std::out_of_range(std::string("remove: No element at position
                        ") + std::to_string(position));
337             } else {
338                 item = item_array[position];
339                 //shift remaining items left
340                 for (size_t i = position + 1; i != size(); ++i) {
341                     item_array[i - 1] = item_array[i];
```

```
342             }
343             --num_items;
344             shrink_if_necessary();
345         }
346         return item;
347     }
348 
349     //-------------------------------------------------
350     /*
351         returns (without removing from the list) the element at the specified
                position.
352     */
353     T item_at(size_t position) const {
354         if (position >= size()) {
355             throw std::out_of_range(std::string("item_at: No element at position
                    ") + std::to_string(position));
356         }
357         return operator[](position);
358     }
359 
360     //-------------------------------------------------
361     /*
362         returns true IFF the list contains no elements.
363     */
364     bool is_empty() const {
365         return size() == 0;
366     }
367 
368     //-------------------------------------------------
369     /*
370         returns the number of elements in the list.
371     */
372     size_t size() const {
373         return num_items;
374     }
375 
376     //-------------------------------------------------
377     /*
378         removes all elements from the list.
379     */
380     void clear() {
381         //no reason to do memory deallocation here, just overwrite the old items
                later and save
382         //deallocation for the deconstructor
383         num_items = 0;
384     }
385 
386     //-------------------------------------------------
387     /*
388         returns true IFF one of the elements of the list matches the specified
                element.
389     */
```

```
390         bool contains(const T& element,
391               bool equals(const T& a, const T& b)) const {
392             bool element_in_list = false;
393             const_iterator fin = end();
394             for (const_iterator iter = begin(); iter != fin; ++iter) {
395                 if (equals(*iter, element)) {
396                     element_in_list = true;
397                     break;
398                 }
399             }
400             return element_in_list;
401         }
402 
403         //-------------------------------------------------
404         /*
405             If the list is empty, inserts "<empty list>" into the ostream;
406             otherwise, inserts, enclosed in square brackets, the list's elements,
407             separated by commas, in sequential order.
408         */
409         std::ostream& print(std::ostream& out) const {
410             if (is_empty()) {
411                 out << "<empty list>";
412             } else {
413                 out << "[";
414                 const_iterator start = begin();
415                 const_iterator fin = end();
416                 for (const_iterator iter = start; iter != fin; ++iter) {
417                     if (iter != start)
418                         out << ",";
419                     out << *iter;
420                 }
421                 out << "]";
422             }
423             return out;
424         }
425     protected:
426         bool validate_internal_integrity() {
427             //todo: fill this in
428             return true;
429         }
430     };
431 } // end namespace cop3530
432 
433 #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     }
129
130 public:
131     //-------------------------------------------------
132     // iterators
133     //-------------------------------------------------
134     class CDAL_Iter: public std::iterator<std::forward_iterator_tag, T> {
135     private:
136         Node* here_container;
137         size_t here_index;
138     public:
139         typedef std::ptrdiff_t difference_type;
140         typedef T& reference;
141         typedef T* pointer;
142         typedef std::forward_iterator_tag iterator_category;
143         typedef T value_type;
144         typedef CDAL_Iter self_type;
145         typedef CDAL_Iter& self_reference;
146
```

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

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

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

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

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

```
390             return remove(size() - 1);
391         }
392 
393         //-------------------------------------------------
394         /*
395             removes and returns the the element at the specified position,
396             shifting the subsequent elements one position to the left.
397         */
398         T remove(size_t position) {
399             T old_item;
400             if (position >= size()) {
401                 throw std::out_of_range(std::string("remove: No element at position
                        ") + std::to_string(position));
402             } else {
403                 ItemLoc loc = loc_from_pos(position);
404                 //shift remaining items to the left
405                 Node* n = loc.node;
406                 old_item = loc.item_ref;
407                 for (size_t i = position; i != num_items; ++i) {
408                     size_t curr_array_index = i % array_size;
409                     size_t next_array_index = (i + 1) % array_size;
410                     T& curr_item = n->item_array[curr_array_index];
411                     if ( next_array_index == 0 ) {
412                         n = n->next;
413                     }
414                     T& next_item = n->item_array[next_array_index];
415                     std::swap(curr_item, next_item);
416                 }
417                 --num_items;
418                 shrink_if_necessary();
419             }
420             return old_item;
421         }
422 
423         //-------------------------------------------------
424         /*
425             returns (without removing from the list) the element at the specified
                    position.
426         */
427         T item_at(size_t position) const {
428             if (position >= size()) {
429                 throw std::out_of_range(std::string("item_at: No element at position
                        ") + std::to_string(position));
430             }
431             return loc_from_pos(position).item_ref;
432         }
433 
434         //-------------------------------------------------
435         /*
436             returns true IFF the list contains no elements.
437         */
438         bool is_empty() const {
```

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

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