Project 1 Deliverable

Paul Nickerson

November 25, 2014

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

SSLL

SSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

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

iterator end()

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

const_iterator begin() const

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

const_iterator end() const

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

T& operator

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

const T& operator const

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

SSLL(const SSLL& src)

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

SSLL& operator=(const SSLL& src)

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

T replace(const T& element, size_t position)

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

void insert(const T& element, size_t position)

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

void push_front(const T& element)

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

void push_back(const T& element)

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

T pop_front()

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

T pop_back()

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

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

T remove(size_t position)

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

T item_at(size_t position) const

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

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

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

void clear()

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

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

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

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

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

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

Iterator Methods

explicit SSLL_Iter(Node* start)

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

SSLL_Iter(const SSLL_Iter& src)

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

reference operator*() const

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

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

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

bool operator!=(const self_type& rhs) const

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

Const Iterator Methods

explicit SSLL_Const_Iter(Node* start)

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

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

SSLL_Const_Iter(const SSLL_Const_Iter& src)

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

reference operator*() const

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

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

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

bool operator!=(const self_type& rhs) const

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

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

PSLL

PSLL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

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

iterator end()

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

const_iterator begin() const

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

const_iterator end() const

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

T& operator

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

const T& operator const

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

PSLL()

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

PSLL(const PSLL& src)

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

PSLL& operator=(const PSLL& src)

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

T replace(const T& element, size_t position)

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

void insert(const T& element, size_t position)

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

void push_front(const T& element)

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

void push_back(const T& element)

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

T pop_front()

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

T pop_back()

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

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

T remove(size_t position)

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

T item_at(size_t position) const

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

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

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

void clear()

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

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

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

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

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

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

Iterator Methods

explicit PSLL_Iter(Node* start)

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

PSLL_Iter(const PSLL_Iter& src)

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

reference operator*() const

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

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

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

bool operator!=(const self_type& rhs) const

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

Const Iterator Methods

explicit PSLL_Const_Iter(Node* start)

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

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

PSLL_Const_Iter(const PSLL_Const_Iter& src)

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

reference operator*() const

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

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

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

bool operator!=(const self_type& rhs) const

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

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

SDAL

SDAL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

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

iterator end()

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

const_iterator begin() const

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

const_iterator end() const

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

T& operator

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

const T& operator const

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

SDAL(size_t num_nodes_to_preallocate = 50)

- Default constructor - takes a parameter which defines the initial array capacity

SDAL(const SDAL& src)

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

SDAL& operator=(const SDAL& src)

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

void embiggen_if_necessary()

- Called whenever we attempt to increase the list size

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

void shrink_if_necessary()

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

T replace(const T& element, size_t position)

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

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- Calls embiggen_if_necessary() to ensure we have space to insert the new item
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
- Providing a position greater than the current list size should throw an out-of-range error

void push_front(const T& element)

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

void push_back(const T& element)

- Inserts a new item to the back of the list calling insert() with the position defined as one past the last stored item

- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

T pop_front()

- Wrapper for remove(0)
- Removes the node at item_array[0] and returns its stored item
- If the list is empty then throw an out-of-range error
- It would be an error if, after popping, size() returned anything besides the old value returned from size() minus one

T pop_back()

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

T remove(size_t position)

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

T item_at(size_t position) const

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

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array

void clear()

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

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

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

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

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

Iterator Methods

explicit SDAL_Iter(T* item_array, T* end_ptr)

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

SDAL_Iter(const SDAL_Iter& src)

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

reference operator*() const

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

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

- Returns true IIF the current and end iter pointers match between current instance and rhs, otherwise returns false

bool operator!=(const self_type& rhs) const

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

Const Iterator Methods

explicit SDAL_Const_Iter(T* item_array, T* end_ptr)

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

SDAL_Const_Iter(const SDAL_Const_Iter& src)

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

reference operator*() const

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

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

- Returns true IIF the current and end iter pointers match between current instance and rhs, otherwise returns false

bool operator!=(const self_type& rhs) const

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

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

CDAL

CDAL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

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

iterator end()

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

const_iterator begin() const

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

const_iterator end() const

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

T& operator

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

const T& operator const

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

CDAL()

- Default constructor - initializes the class by allocating head/tail dummy nodes, then adding an initial node

CDAL(const CDAL& src)

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

CDAL& operator=(const CDAL& src)

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

void embiggen_if_necessary()

- Called whenever we attempt to increase the list size
- If each array slot in every link is filled and we want to add a new item, allocate and append a new link by transforming the tail node into a usable item array container that points to a freshly-allocated tail node
- If we fail to allocate nodes, throw a bad_alloc exception

void shrink_if_necessary()

- Called whenever we attempt to decrease the list size
- Because we don't want the list to waste too much memory, whenever more than half of the arrays are unused (they would all be at the end of the chain), we deallocate half the arrays by traversing to the last node to keep, then dropping each subsequent node until we reach the tail

T replace(const T& element, size_t position)

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

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- Calls embiggen_if_necessary() to ensure we have space to insert the new item
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
- Providing a position greater than the current list size should throw an out-of-range error

void push_front(const T& element)

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

void push_back(const T& element)

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

T pop_front()

- Wrapper for remove(0)
- Removes the node at item_array[0] and returns its stored item
- If the list is empty then throw an out-of-range error
- It would be an error if, after popping, size() returned anything besides the old value returned from size() minus one

T pop_back()

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

T remove(size_t position)

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

T item_at(size_t position) const

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

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

- Returns value of the counter which tracks the number of items stored in the array

void clear()

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

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

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

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

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

Iterator Methods

CDAL_Iter(ItemLoc const& here)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at the node and array index described by the here parameter
- Neither item_array nor end_ptr may be null
- end_ptr must be greater than or equal to item_array

CDAL_Iter(const CDAL_Iter& src)

- Copy constructor - sets the current iterator position to the node and array index described by src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

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

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

- Returns true IIF the current and end iter pointers match between current instance and rhs, otherwise returns false

bool operator!=(const self_type& rhs) const

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

Const Iterator Methods

CDAL_Iter(ItemLoc const& here)

- Explicit constructor for an iterator which, when dereferenced, returns an immutable reference to the item held at the node and array index described by the here parameter

CDAL_Const_Iter(const CDAL_Const_Iter& src)

- Copy constructor - sets the current iterator position to the node and array index described by src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

reference operator*() const

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

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

- Returns true IIF the current and end iter pointers match between current instance and rhs, otherwise returns false

bool operator!=(const self_type& rhs) const

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

CDAL Informal Documentation

Paul Nickerson

Something here

this is a test hello world

Something here

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-: no
* 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-: no
* 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-: no
* 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-: no
* 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* allocate_new_node() {
53          Node* n;
54          try {
55              n = new Node();
56          } catch (std::bad_alloc& ba) {
57              std::cerr << "allocate_new_node(): failed to allocate memory for new
                    node" << std::endl;
58              throw std::bad_alloc();
59          }
60          return n;
61      }
62      Node* design_new_node(const T& element, Node* next = nullptr, bool dummy =
            false) {
63          Node* n = allocate_new_node();
64          n->is_dummy = dummy;
65          n->item = element;
66          n->next = next;
67          return n;
68      }
69      Node* design_new_node(Node* next = nullptr, bool dummy = false) {
70          Node* n = allocate_new_node();
71          n->is_dummy = dummy;
72          n->next = next;
73          return n;
74      }
75      void init() {
76          num_items = 0;
77          try {
78              tail = design_new_node(nullptr, true);
79              head = design_new_node(tail, true);
80          } catch (std::bad_alloc& ba) {
81              std::cerr << "init(): failed to allocate memory for head/tail nodes"
                    << std::endl;
82              throw std::bad_alloc();
83          }
84      }
85      //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)
86      void insert_node_after(Node* existing_node, Node* new_node) {
87          existing_node->next = new_node;
88          ++num_items;
89      }
90      //destroys the subsequent node and returns its item
91      T remove_item_after(Node* preceeding_node) {
92          Node* removed_node = preceeding_node->next;
93          T item = removed_node->item;
94          preceeding_node->next = removed_node->next;
```

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

```
145             }
146             self_reference operator=( const self_type& src ) {
147                 if (&src == this)
148                     return *this;
149                 here = src.here;
150                 if (*this != src)
151                     throw std::runtime_error("SSLL_Iter: copy assignment failed");
152                 return *this;
153             }
154             self_reference operator++() { // preincrement
155                 if (here->is_dummy)
156                     throw std::out_of_range("SSLL_Iter: Can't traverse past the end
                            of the list");
157                 here = here->next;
158                 return *this;
159             }
160             self_type operator++(int) { // postincrement
161                 self_type t(*this); //save state
162                 operator++(); //apply increment
163                 return t; //return state held before increment
164             }
165             bool operator==(const self_type& rhs) const {
166                 return rhs.here == here;
167             }
168             bool operator!=(const self_type& rhs) const {
169                 return ! operator==(rhs);
170             }
171         };
172
173         class SSLL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
174         {
175         public:
176             // inheriting from std::iterator<std::forward_iterator_tag, T>
177             // automagically sets up these typedefs...
178             typedef T value_type;
179             typedef std::ptrdiff_t difference_type;
180             typedef const T& reference;
181             typedef const T* pointer;
182             typedef std::forward_iterator_tag iterator_category;
183
184             // but not these typedefs...
185             typedef SSLL_Const_Iter self_type;
186             typedef SSLL_Const_Iter& self_reference;
187
188         private:
189             const Node* here;
190
191         public:
192             explicit SSLL_Const_Iter(Node* start) : here(start) {
193                 if (start == nullptr)
194                     throw std::runtime_error("SSLL_Const_Iter: start cannot be null");
195             }
```

```
196             SSLL_Const_Iter(const SSLL_Const_Iter& src) : here(src.here) {}
197             SSLL_Const_Iter(const SSLL_Iter& src) : here(src.here) {}
198
199             reference operator*() const {
200                 if (here->is_dummy)
201                     throw std::out_of_range("SSLL_Const_Iter: can't dereference end
                            position");
202                 return here->item;
203             }
204             pointer operator->() const {
205                 return & this->operator*();
206             }
207             self_reference operator=( const self_type& src ) {
208                 if (&src == this)
209                     return *this;
210                 here = src.here;
211                 if (*this != src)
212                     throw std::runtime_error("SSLL_Const_Iter: copy assignment
                            failed");
213                 return *this;
214             }
215             self_reference operator++() { // preincrement
216                 if (here->is_dummy)
217                     throw std::out_of_range("SSLL_Const_Iter: Can't traverse past the
                            end of the list");
218                 here = here->next;
219                 return *this;
220             }
221             self_type operator++(int) { // postincrement
222                 self_type t(*this); //save state
223                 operator++(); //apply increment
224                 return t; //return state held before increment
225             }
226             bool operator==(const self_type& rhs) const {
227                 return rhs.here == here;
228             }
229             bool operator!=(const self_type& rhs) const {
230                 return ! operator==(rhs);
231             }
232         };
233
234         //-------------------------------------------------
235         // types
236         //-------------------------------------------------
237         typedef T value_type;
238         typedef SSLL_Iter iterator;
239         typedef SSLL_Const_Iter const_iterator;
240
241         iterator begin() { return SSLL_Iter(head->next); }
242         iterator end() { return SSLL_Iter(tail); }
243
244         const_iterator begin() const { return SSLL_Const_Iter(head->next); }
```

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

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

```
338                 std::cerr << "insert(): failed to allocate memory for new node"
                        << std::endl;
339                 throw std::bad_alloc();
340             }
341             insert_node_after(node_before_position, new_node);
342         }
343     }
344
345     /*
346         prepends the specified element to the list.
347     */
348     void push_front(const T& element) {
349         insert(element, 0);
350     }
351
352     //-------------------------------------------------
353     /*
354         appends the specified element to the list.
355     */
356     void push_back(const T& element) {
357         Node* new_tail;
358         try {
359             new_tail = design_new_node(nullptr, true);
360         } catch (std::bad_alloc& ba) {
361             std::cerr << "push_back(): failed to allocate memory for new tail"
                        << std::endl;
362             throw std::bad_alloc();
363         }
364         insert_node_after(tail, new_tail);
365         //transform the current tail node from a dummy to a real node holding
                element
366         tail->is_dummy = false;
367         tail->item = element;
368         tail->next = new_tail;
369         tail = tail->next;
370     }
371
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         if (head->next == tail) {
380             throw std::runtime_error("pop_front: head->next == tail, but list
                        says it's not empty (corrupt state)");
381         }
382         return remove_item_after(head);
383     }
384
385     //-------------------------------------------------
```

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

```
432         //-------------------------------------------------
433         /*
434             returns true IFF the list contains no elements.
435         */
436         bool is_empty() const {
437             return size() == 0;
438         }
439 
440         //-------------------------------------------------
441         /*
442             returns the number of elements in the list.
443         */
444         size_t size() const {
445             if (num_items == 0 && head->next != tail) {
446                 throw std::runtime_error("size: head->next != tail, but list says
                        it's empty (corrupt state)");
447             } else if (num_items > 0 && head->next == tail) {
448                 throw std::runtime_error("size: head->next == tail, but list says
                        it's not empty (corrupt state)");
449             }
450             return num_items;
451         }
452 
453         //-------------------------------------------------
454         /*
455             removes all elements from the list.
456         */
457         void clear() {
458             while ( ! is_empty()) {
459                 pop_front();
460             }
461         }
462 
463         //-------------------------------------------------
464         /*
465             returns true IFF one of the elements of the list matches the specified
                element.
466         */
467         bool contains(const T& element,
468                bool equals(const T& a, const T& b)) const {
469             bool element_in_list = false;
470             const_iterator fin = end();
471             for (const_iterator iter = begin(); iter != fin; ++iter) {
472                 if (equals(*iter, element)) {
473                     element_in_list = true;
474                     break;
475                 }
476             }
477             return element_in_list;
478         }
479 
480         //-------------------------------------------------
```

```
481         /*
482             If the list is empty, inserts "<empty list>" into the ostream;
483             otherwise, inserts, enclosed in square brackets, the list's elements,
484             separated by commas, in sequential order.
485         */
486         std::ostream& print(std::ostream& out) const {
487             if (is_empty()) {
488                 out << "<empty list>";
489             } else {
490                 out << "[";
491                 const_iterator start = begin();
492                 const_iterator fin = end();
493                 for (const_iterator iter = start; iter != fin; ++iter) {
494                     if (iter != start)
495                         out << ",";
496                     out << *iter;
497                 }
498                 out << "]";
499             }
500             return out;
501         }
502     protected:
503         bool validate_internal_integrity() {
504             //todo: fill this in
505             return true;
506         }
507     }; //end class SSLL
508 } // end namespace cop3530
509 #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-: no
* 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-: no
* 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-: no
* 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-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

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

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

My UNIT TESTS run correctly on the OpenBSD VM: yes

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

I affirm that I am the sole author of this Pool-using Singly-Linked List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

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

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

psll/source/PSLL.h

PSLL.h

```
#ifndef _PSLL_H_
#define _PSLL_H_

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

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

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

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

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

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

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

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

```
291             }
292             return node_at(i)->item;
293         }
294 
295         //-------------------------------------------------
296         // Constructors/destructor/assignment operator
297         //-------------------------------------------------
298 
299         PSLL() {
300             init();
301         }
302         //-------------------------------------------------
303         //copy constructor
304         PSLL(const PSLL& src) {
305             init();
306             copy_constructor(src);
307         }
308 
309         //-------------------------------------------------
310         //destructor
311         ~PSLL() {
312             // safely dispose of this PSLL's contents
313             clear();
314         }
315 
316         //-------------------------------------------------
317         //copy assignment constructor
318         PSLL& operator=(const PSLL& src) {
319             if (&src == this) // check for self-assignment
320                 return *this;   // do nothing
321             // safely dispose of this PSLL's contents
322             clear();
323             // populate this PSLL with copies of the other PSLL's contents
324             copy_constructor(src);
325             return *this;
326         }
327 
328         //-------------------------------------------------
329         // member functions
330         //-------------------------------------------------
331 
332         /*
333             replaces the existing element at the specified position with the
                    specified element and
334             returns the original element.
335         */
336         T replace(const T& element, size_t position) {
337             T old_item;
338             if (position >= size()) {
339                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
340             } else {
```

```
341             //we are guaranteed to be at a non-dummy item now because of the
                    above if statement
342             Node* iter = node_at(position);
343             old_item = iter->item;
344             iter->item = element;
345         }
346         return old_item;
347     }
348 
349     //-------------------------------------------------
350     /*
351         adds the specified element to the list at the specified position,
                shifting the element
352         originally at that and those in subsequent positions one position to the
                right.
353     */
354     void insert(const T& element, size_t position) {
355         if (position > size()) {
356             throw std::out_of_range(std::string("insert: Position is outside of
                    the list: ") + std::to_string(position));
357         } else if (position == size()) {
358             //special O(1) case
359             push_back(element);
360         } else {
361             //node_before_position is guaranteed to point to a valid node
                    because we use a dummy head node
362             Node* node_before_position = node_before(position);
363             Node* node_at_position = node_before_position->next;
364             Node* new_node;
365             try {
366                 new_node = design_new_node(element, node_at_position);
367             } catch (std::bad_alloc& ba) {
368                 std::cerr << "insert(): failed to allocate memory for new node"
                        << std::endl;
369                 throw std::bad_alloc();
370             }
371             insert_node_after(node_before_position, new_node,
                    num_main_list_items);
372         }
373     }
374 
375     //-------------------------------------------------
376     //Note to self: use reference here because we receive the original object
            instance,
377     //then copy it into n->item so we have it if the original element goes out
            of scope
378     /*
379         prepends the specified element to the list.
380     */
381     void push_front(const T& element) {
382         insert(element, 0);
383     }
```

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

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

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

```
525                 out << "[";
526                 const_iterator start = begin();
527                 const_iterator fin = end();
528                 for (const_iterator iter = start; iter != fin; ++iter) {
529                     if (iter != start)
530                         out << ",";
531                     out << *iter;
532                 }
533                 out << "]";
534             }
535             return out;
536         }
537     protected:
538         bool validate_internal_integrity() {
539             //todo: fill this in
540             return true;
541         }
542     }; //end class PSLL
543 } // end namespace cop3530
544 #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-: no
* 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-: no
* 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-: no
* 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-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

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

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

My UNIT TESTS run correctly on the OpenBSD VM: yes

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

I affirm that I am the sole author of this Simple Dynamic Array-based List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

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

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

sdal/source/SDAL.h

SDAL.h

```
#ifndef _SDAL_H_
#define _SDAL_H_

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

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

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

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

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

```
141             return *this;
142         }
143         self_reference operator++() { // preincrement
144             if (iter == end_iter)
145                 throw std::out_of_range("SDAL_Iter: Can't traverse past the end
                        of the list");
146             ++iter;
147             return *this;
148         }
149         self_type operator++(int) { // postincrement
150             self_type t(*this); //save state
151             operator++(); //apply increment
152             return t; //return state held before increment
153         }
154         bool operator==(const self_type& rhs) const {
155             return rhs.iter == iter && rhs.end_iter == end_iter;
156         }
157         bool operator!=(const self_type& rhs) const {
158             return ! operator==(rhs);
159         }
160     };
161
162     class SDAL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
163     {
164     public:
165         // inheriting from std::iterator<std::forward_iterator_tag, T>
166         // automagically sets up these typedefs...
167         typedef T value_type;
168         typedef std::ptrdiff_t difference_type;
169         typedef const T& reference;
170         typedef const T* pointer;
171         typedef std::forward_iterator_tag iterator_category;
172
173         // but not these typedefs...
174         typedef SDAL_Const_Iter self_type;
175         typedef SDAL_Const_Iter& self_reference;
176     private:
177         const T* iter;
178         const T* end_iter;
179     public:
180         explicit SDAL_Const_Iter(T* item_array, T* end_ptr): iter(item_array),
                     end_iter(end_ptr) {
181             if (item_array == nullptr)
182                 throw std::runtime_error("SDAL_Const_Iter: item_array cannot be
                        null");
183             if (end_ptr == nullptr)
184                 throw std::runtime_error("SDAL_Const_Iter: end_ptr cannot be
                        null");
185             if (item_array > end_ptr)
186                 throw std::runtime_error("SDAL_Const_Iter: item_array pointer
                        cannot be past end_ptr");
187         }
```

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

```
234         iterator end() { return SDAL_Iter(item_array + num_items, item_array +
                num_items); }
235
236         const_iterator begin() const { return SDAL_Const_Iter(item_array,
                item_array + num_items); }
237         const_iterator end() const { return SDAL_Const_Iter(item_array + num_items,
                item_array + num_items); }
238
239         //-------------------------------------------------
240         // operators
241         //-------------------------------------------------
242         T& operator[](size_t i) {
243             if (i >= size()) {
244                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
245             }
246             return item_array[i];
247         }
248
249         const T& operator[](size_t i) const {
250             if (i >= size()) {
251                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
252             }
253             return item_array[i];
254         }
255
256         //-------------------------------------------------
257         // Constructors/destructor/assignment operator
258         //-------------------------------------------------
259
260         SDAL(size_t num_nodes_to_preallocate = 50) {
261             init(num_nodes_to_preallocate);
262         }
263
264         //-------------------------------------------------
265         //copy constructor
266         SDAL(const SDAL& src): SDAL(src.array_size) {
267             init(src.array_size);
268             copy_constructor(src);
269         }
270
271         //-------------------------------------------------
272         //destructor
273         ~SDAL() {
274             // safely dispose of this SDAL's contents
275             delete[] item_array;
276         }
277
278         //-------------------------------------------------
279         //copy assignment constructor
280         SDAL& operator=(const SDAL& src) {
```

```
281             if (&src == this) // check for self-assignment
282                 return *this;   // do nothing
283             delete[] item_array;
284             init(src.array_size);
285             copy_constructor(src);
286             return *this;
287         }
288 
289         //-------------------------------------------------
290         // member functions
291         //-------------------------------------------------
292 
293         /*
294             replaces the existing element at the specified position with the
                    specified element and
295             returns the original element.
296         */
297         T replace(const T& element, size_t position) {
298             T old_item;
299             if (position >= size()) {
300                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
301             } else {
302                 old_item = item_array[position];
303                 item_array[position] = element;
304             }
305             return old_item;
306         }
307 
308         //-------------------------------------------------
309         /*
310             adds the specified element to the list at the specified position,
                    shifting the element
311             originally at that and those in subsequent positions one position to the
                    right.
312         */
313         void insert(const T& element, size_t position) {
314             if (position > size()) {
315                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
316             } else {
317                 embiggen_if_necessary();
318                 //shift remaining items right
319                 for (size_t i = size(); i != position; --i) {
320                     item_array[i] = item_array[i - 1];
321                 }
322                 item_array[position] = element;
323                 ++num_items;
324             }
325         }
326 
327         //-------------------------------------------------
```

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

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

```
424             //deallocation for the deconstructor
425             num_items = 0;
426         }
427 
428         //-------------------------------------------------
429         /*
430             returns true IFF one of the elements of the list matches the specified
                element.
431         */
432         bool contains(const T& element,
433               bool equals(const T& a, const T& b)) const {
434             bool element_in_list = false;
435             const_iterator fin = end();
436             for (const_iterator iter = begin(); iter != fin; ++iter) {
437                 if (equals(*iter, element)) {
438                     element_in_list = true;
439                     break;
440                 }
441             }
442             return element_in_list;
443         }
444 
445         //-------------------------------------------------
446         /*
447             If the list is empty, inserts "<empty list>" into the ostream;
448             otherwise, inserts, enclosed in square brackets, the list's elements,
449             separated by commas, in sequential order.
450         */
451         std::ostream& print(std::ostream& out) const {
452             if (is_empty()) {
453                 out << "<empty list>";
454             } else {
455                 out << "[";
456                 const_iterator start = begin();
457                 const_iterator fin = end();
458                 for (const_iterator iter = start; iter != fin; ++iter) {
459                     if (iter != start)
460                         out << ",";
461                     out << *iter;
462                 }
463                 out << "]";
464             }
465             return out;
466         }
467     protected:
468         bool validate_internal_integrity() {
469             //todo: fill this in
470             return true;
471         }
472     };
473 } // end namespace cop3530
474 
```

```
475 #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-: no
* 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-: no
* 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-: no
* 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-: no
* 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)
```

```
47              return head;
48          else
49              return node_at(position - 1);
50      }
51
52      ItemLoc loc_from_pos(size_t position) const {
53          size_t node_position = floor(position / array_size);
54          Node* n = node_at(node_position);
55          size_t array_index = position % array_size;
56          ItemLoc loc {n, array_index, n->item_array[array_index]};
57          return loc;
58      }
59
60      Node* design_new_node(Node* next = nullptr, bool dummy = false) const {
61          Node* n;
62          try {
63              n = new Node();
64          } catch (std::bad_alloc& ba) {
65              std::cerr << "design_new_node(): failed to allocate memory for new
                    node" << std::endl;
66              throw std::bad_alloc();
67          }
68          n->is_dummy = dummy;
69          try {
70              n->item_array = new T[array_size];
71          } catch (std::bad_alloc& ba) {
72              std::cerr << "design_new_node(): failed to allocate memory for item
                    array" << std::endl;
73              throw std::bad_alloc();
74          }
75          n->next = next;
76          return n;
77      }
78
79      void init() {
80          num_items = 0;
81          num_available_nodes = 0;
82          tail = design_new_node(nullptr, true);
83          head = design_new_node(tail, true);
84      }
85
86      void free_node(Node* n) {
87          delete[] n->item_array;
88          delete n;
89      }
90
91      void drop_node_after(Node* n) {
92          assert(n->next != tail);
93          Node* removed_node = n->next;
94          n->next = removed_node->next;
95          free_node(removed_node);
96          --num_available_nodes;
```

```
97          }
98
99          size_t num_used_nodes() {
100             return ceil(size() / array_size);
101         }
102
103         void embiggen_if_necessary() {
104             //embiggen is a perfectly cromulent word
105             /*
106                 If each array slot in every link is filled and we want to add a new
                        item, allocate and append a new link
107             */
108             if (size() == num_available_nodes * array_size) {
109                 //transform tail into a regular node and append a new tail
110                 Node* n = tail;
111                 n->is_dummy = false;
112                 tail = n->next = design_new_node(nullptr, true);
113                 ++num_available_nodes;
114                 ++embiggen_counter;
115             }
116         }
117
118         void shrink_if_necessary() {
119             /*
120                 Because we don't want the list to waste too much memory, whenever
                        the more than half of the arrays
121                 are unused (they would all be at the end of the chain), deallocate
                        half the unused arrays.
122             */
123             size_t used = num_used_nodes();
124             size_t num_unused_nodes = num_available_nodes - used;
125             if (num_unused_nodes > used) {
126                 size_t nodes_to_keep = used + ceil(num_unused_nodes * 0.5);
127                 Node* last_node = node_before(nodes_to_keep);
128                 while (last_node->next != tail) {
129                     drop_node_after(last_node);
130                 }
131                 ++shrink_counter;
132             }
133         }
134         void copy_constructor(const CDAL& src) {
135             const_iterator fin = src.end();
136             for (const_iterator iter = src.begin(); iter != fin; ++iter) {
137                 push_back(*iter);
138             }
139             if ( ! src.size() == size())
140                 throw std::runtime_error("copy_constructor: Copying failed - sizes
                        don't match up");
141         }
142
143     public:
144         //-------------------------------------------------
```

```
145         // iterators
146         //-------------------------------------------------
147         class CDAL_Const_Iter;
148         class CDAL_Iter: public std::iterator<std::forward_iterator_tag, T> {
149             friend class CDAL_Const_Iter;
150         private:
151             Node* curr_node;
152             size_t curr_array_index;
153             Node* fin_node;
154             size_t fin_array_index;
155         public:
156             typedef std::ptrdiff_t difference_type;
157             typedef T& reference;
158             typedef T* pointer;
159             typedef std::forward_iterator_tag iterator_category;
160             typedef T value_type;
161             typedef CDAL_Iter self_type;
162             typedef CDAL_Iter& self_reference;
163 
164             //need copy constructor/assigner to make this a first class ADT (doesn't
                    hold pointers that need freeing)
165             CDAL_Iter(ItemLoc const& here, ItemLoc const& fin):
166                 curr_node(here.node),
167                 curr_array_index(here.array_index),
168                 fin_node(fin.node),
169                 fin_array_index(fin.array_index)
170             {}
171             CDAL_Iter(const self_type& src):
172                 curr_node(src.curr_node),
173                 curr_array_index(src.curr_array_index),
174                 fin_node(src.fin_node),
175                 fin_array_index(src.fin_array_index)
176             {}
177             self_reference operator=(const self_type& rhs) {
178                 //copy assigner
179                 if (&rhs == this) return *this;
180                 curr_node = rhs.curr_node;
181                 curr_array_index = rhs.curr_array_index;
182                 fin_node = rhs.fin_node;
183                 fin_array_index = rhs.fin_array_index;
184 
185                 if (*this != rhs)
186                     throw std::runtime_error("CDAL_Iter: copy assignment failed");
187                 return *this;
188             }
189             self_reference operator++() { // preincrement
190                 if (curr_node == fin_node && curr_array_index == fin_array_index)
191                     throw std::out_of_range("CDAL_Iter: Can't traverse past the end
                            of the list");
192                 curr_array_index = (curr_array_index + 1) % array_size;
193                 if (curr_array_index == 0) curr_node = curr_node->next;
194                 return *this;
```

```
195             }
196             self_type operator++(int) { // postincrement
197                 self_type t(*this); //save state
198                 operator++(); //apply increment
199                 return t; //return state held before increment
200             }
201             reference operator*() const {
202                 if (curr_node == fin_node && curr_array_index == fin_array_index)
203                     throw std::out_of_range("SSLL_Iter: can't dereference end
                            position");
204                 return curr_node->item_array[curr_array_index];
205             }
206             pointer operator->() const {
207                 return & this->operator*();
208             }
209             bool operator==(const self_type& rhs) const {
210                 return rhs.curr_node == curr_node
211                        && rhs.curr_array_index == curr_array_index;
212             }
213             bool operator!=(const self_type& rhs) const {
214                 return ! operator==(rhs);
215             }
216         };
217
218         class CDAL_Const_Iter: public std::iterator<std::forward_iterator_tag, T> {
219         private:
220             Node* curr_node;
221             size_t curr_array_index;
222             Node* fin_node;
223             size_t fin_array_index;
224         public:
225             typedef std::ptrdiff_t difference_type;
226             typedef const T& reference;
227             typedef const T* pointer;
228             typedef std::forward_iterator_tag iterator_category;
229             typedef T value_type;
230             typedef CDAL_Const_Iter self_type;
231             typedef CDAL_Const_Iter& self_reference;
232
233             //need copy constructor/assigner to make this a first class ADT (doesn't
                    hold pointers that need freeing)
234             CDAL_Const_Iter(ItemLoc const& here, ItemLoc const& fin):
235                 curr_node(here.node),
236                 curr_array_index(here.array_index),
237                 fin_node(fin.node),
238                 fin_array_index(fin.array_index)
239             {}
240             CDAL_Const_Iter(const self_type& src):
241                 curr_node(src.curr_node),
242                 curr_array_index(src.curr_array_index),
243                 fin_node(src.fin_node),
244                 fin_array_index(src.fin_array_index)
```

```
245             {}
246             CDAL_Const_Iter(const CDAL_Iter& src):
247                 curr_node(src.curr_node),
248                 curr_array_index(src.curr_array_index),
249                 fin_node(src.fin_node),
250                 fin_array_index(src.fin_array_index)
251             {}
252             self_reference operator=(const self_type& rhs) {
253                 //copy assigner
254                 if (&rhs == this) return *this;
255                 curr_node = rhs.curr_node;
256                 curr_array_index = rhs.curr_array_index;
257                 fin_node = rhs.fin_node;
258                 fin_array_index = rhs.fin_array_index;
259
260                 if (*this != rhs)
261                     throw std::runtime_error("CDAL_Const_Iter: copy assignment
                            failed");
262                 return *this;
263             }
264             self_reference operator++() { // preincrement
265                 if (curr_node == fin_node && curr_array_index == fin_array_index)
266                     throw std::out_of_range("CDAL_Const_Iter: Can't traverse past the
                            end of the list");
267                 curr_array_index = (curr_array_index + 1) % array_size;
268                 if (curr_array_index == 0) curr_node = curr_node->next;
269                 return *this;
270             }
271             self_type operator++(int) { // postincrement
272                 self_type t(*this); //save state
273                 operator++(); //apply increment
274                 return t; //return state held before increment
275             }
276             reference operator*() const {
277                 if (curr_node == fin_node && curr_array_index == fin_array_index)
278                     throw std::out_of_range("SSLL_Iter: can't dereference end
                            position");
279                 return curr_node->item_array[curr_array_index];
280             }
281             pointer operator->() const {
282                 return & this->operator*();
283             }
284             bool operator==(const self_type& rhs) const {
285                 return rhs.curr_node == curr_node
286                        && rhs.curr_array_index == curr_array_index;
287             }
288             bool operator!=(const self_type& rhs) const {
289                 return ! operator==(rhs);
290             }
291         };
292
293         //-------------------------------------------------
```

```
294         // types
295         //-----------------------------------------------
296         typedef CDAL_Iter iterator;
297         typedef CDAL_Const_Iter const_iterator;
298         typedef T value_type;
299         //todo: might need to add size_t here and other iterators if they were
                excluded or commented out
300
301         iterator begin() {
302             ItemLoc start_loc = loc_from_pos(0);
303             ItemLoc end_loc = loc_from_pos(size());
304             return iterator(start_loc, end_loc);
305         }
306
307         iterator end() {
308             ItemLoc end_loc = loc_from_pos(size());
309             return iterator(end_loc, end_loc);
310         }
311
312         const_iterator begin() const {
313             ItemLoc start_loc = loc_from_pos(0);
314             ItemLoc end_loc = loc_from_pos(size());
315             return const_iterator(start_loc, end_loc);
316         }
317
318         const_iterator end() const {
319             ItemLoc end_loc = loc_from_pos(size());
320             return const_iterator(end_loc, end_loc);
321         }
322
323         T& operator[](size_t i) {
324             if (i >= size()) {
325                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
326             }
327             return loc_from_pos(i).item_ref;
328         }
329
330         const T& operator[](size_t i) const {
331             if (i >= size()) {
332                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
333             }
334             return loc_from_pos(i).item_ref;
335         }
336
337         //-----------------------------------------------
338         // Constructors/destructor/assignment operator
339         //-----------------------------------------------
340
341         CDAL() {
342             init();
```

```
343             embiggen_if_necessary();
344         }
345         //-------------------------------------------------
346         //copy constructor
347         CDAL(const CDAL& src) {
348             init();
349             copy_constructor(src);
350         }
351 
352         //-------------------------------------------------
353         //destructor
354         ~CDAL() {
355             // safely dispose of this CDAL's contents
356             clear();
357         }
358 
359         //-------------------------------------------------
360         //copy assignment constructor
361         CDAL& operator=(const CDAL& src) {
362             if (&src == this) // check for self-assignment
363                 return *this;   // do nothing
364             // safely dispose of this CDAL's contents
365             // populate this CDAL with copies of the other CDAL's contents
366             clear();
367             init();
368             copy_constructor(src);
369             return *this;
370         }
371 
372         //-------------------------------------------------
373         // member functions
374         //-------------------------------------------------
375 
376         /*
377             replaces the existing element at the specified position with the
                    specified element and
378             returns the original element.
379         */
380         T replace(const T& element, size_t position) {
381             T item = element;
382             if (position >= size()) {
383                 throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
384             } else {
385                 ItemLoc loc = loc_from_pos(position);
386                 std::swap(loc.item_ref, item);
387             }
388             return item;
389         }
390 
391         //-------------------------------------------------
392         /*
```

```
393             adds the specified element to the list at the specified position,
                    shifting the element
394             originally at that and those in subsequent positions one position to the
                    right.
395         */
396         void insert(const T& element, size_t position) {
397             if (position > size()) {
398                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
399             } else {
400                 embiggen_if_necessary();
401                 ItemLoc loc = loc_from_pos(position);
402                 //shift remaining items to the right
403                 T item_to_insert = element;
404                 Node* n = loc.node;
405                 for (size_t i = position; i <= num_items; ++i) {
406                     size_t array_index = i % array_size;
407                     if ( i != position && array_index == 0 ) {
408                         n = n->next;
409                     }
410                     std::swap(item_to_insert, n->item_array[array_index]);
411                 }
412                 ++num_items;
413             }
414         }
415
416         //-------------------------------------------------
417         //Note to self: use reference here because we receive the original object
                instance,
418         //then copy it into n->item so we have it if the original element goes out
                of scope
419         /*
420             prepends the specified element to the list.
421         */
422         void push_front(const T& element) {
423             insert(element, 0);
424         }
425
426         //-------------------------------------------------
427         /*
428             appends the specified element to the list.
429         */
430         void push_back(const T& element) {
431             insert(element, size());
432         }
433
434         //-------------------------------------------------
435         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
436         //of that so the client still has a valid instance if our destructor is
                called
437         /*
```

```
438             removes and returns the element at the list's head.
439         */
440         T pop_front() {
441             if (is_empty()) {
442                 throw std::out_of_range("pop_front: Can't pop: list is empty");
443             }
444             return remove(0);
445         }
446 
447         //-------------------------------------------------
448         /*
449             removes and returns the element at the list's tail.
450         */
451         T pop_back() {
452             if (is_empty()) {
453                 throw std::out_of_range("pop_back: Can't pop: list is empty");
454             }
455             return remove(size() - 1);
456         }
457 
458         //-------------------------------------------------
459         /*
460             removes and returns the the element at the specified position,
461             shifting the subsequent elements one position to the left.
462         */
463         T remove(size_t position) {
464             T old_item;
465             if (position >= size()) {
466                 throw std::out_of_range(std::string("remove: No element at position
                        ") + std::to_string(position));
467             } else {
468                 ItemLoc loc = loc_from_pos(position);
469                 //shift remaining items to the left
470                 Node* n = loc.node;
471                 old_item = loc.item_ref;
472                 for (size_t i = position; i != num_items; ++i) {
473                     size_t curr_array_index = i % array_size;
474                     size_t next_array_index = (i + 1) % array_size;
475                     T& curr_item = n->item_array[curr_array_index];
476                     if ( next_array_index == 0 ) {
477                         n = n->next;
478                     }
479                     T& next_item = n->item_array[next_array_index];
480                     std::swap(curr_item, next_item);
481                 }
482                 --num_items;
483                 shrink_if_necessary();
484             }
485             return old_item;
486         }
487 
488         //-------------------------------------------------
```

```
489         /*
490             returns (without removing from the list) the element at the specified
                position.
491         */
492         T item_at(size_t position) const {
493             if (position >= size()) {
494                 throw std::out_of_range(std::string("item_at: No element at position
                        ") + std::to_string(position));
495             }
496             return operator[](position);
497         }
498 
499         //-------------------------------------------------
500         /*
501             returns true IFF the list contains no elements.
502         */
503         bool is_empty() const {
504             return size() == 0;
505         }
506 
507         //-------------------------------------------------
508         /*
509             returns the number of elements in the list.
510         */
511         size_t size() const {
512             return num_items;
513         }
514 
515         //-------------------------------------------------
516         /*
517             removes all elements from the list.
518         */
519         void clear() {
520             while (head->next != tail) {
521                 drop_node_after(head);
522             }
523             num_items = 0;
524         }
525         //-------------------------------------------------
526         /*
527             returns true IFF one of the elements of the list matches the specified
                element.
528         */
529         bool contains(const T& element,
530               bool equals(const T& a, const T& b)) const {
531             bool element_in_list = false;
532             const_iterator fin = end();
533             for (const_iterator iter = begin(); iter != fin; ++iter) {
534                 if (equals(*iter, element)) {
535                     element_in_list = true;
536                     break;
537                 }
```

```
538             }
539             return element_in_list;
540         }
541 
542         //-------------------------------------------------
543         /*
544             If the list is empty, inserts "<empty list>" into the ostream;
545             otherwise, inserts, enclosed in square brackets, the list's elements,
546             separated by commas, in sequential order.
547         */
548         std::ostream& print(std::ostream& out) const {
549             if (is_empty()) {
550                 out << "<empty list>";
551             } else {
552                 out << "[";
553                 const_iterator start = begin();
554                 const_iterator fin = end();
555                 for (const_iterator iter = start; iter != fin; ++iter) {
556                     if (iter != start)
557                         out << ",";
558                     out << *iter;
559                 }
560                 out << "]";
561             }
562             return out;
563         }
564     }; //end class CDAL
565 } // end namespace cop3530
566 #endif // _CDAL_H_
```