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)
                return head;
            else
                return node_at(position - 1);
        }
        Node* allocate_new_node() {
            Node* n;
            try {
                n = new Node();
            } catch (std::bad_alloc& ba) {
                std::cerr << "allocate_new_node(): failed to allocate memory for new node" << std::endl;
                throw std::bad_alloc();
            }
            return n;
        }
        Node* design_new_node(const T& element, Node* next = nullptr, bool dummy = false) {
            Node* n = allocate_new_node();
            n->is_dummy = dummy;
            n->item = element;
            n->next = next;
            return n;
        }
        Node* design_new_node(Node* next = nullptr, bool dummy = false) {
            Node* n = allocate_new_node();
            n->is_dummy = dummy;
            n->next = next;
            return n;
        }
        void init() {
            num_items = 0;
            try {
                tail = design_new_node(nullptr, true);
                head = design_new_node(tail, true);
            } catch (std::bad_alloc& ba) {
                std::cerr << "init(): failed to allocate memory for head/tail nodes" << std::endl;
                throw std::bad_alloc();
            }
        }
        //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)
        void insert_node_after(Node* existing_node, Node* new_node) {
            existing_node->next = new_node;
            ++num_items;
        }
        //destroys the subsequent node and returns its item
        T remove_item_after(Node* preceeding_node) {
            Node* removed_node = preceeding_node->next;
            T item = removed_node->item;
            preceeding_node->next = removed_node->next;
```

```
            delete removed_node;
            --num_items;
            return item;
        }
        void copy_constructor(const SSLL& src) {
            const_iterator fin = src.end();
            for (const_iterator iter = src.begin(); iter != fin; ++iter) {
                push_back(*iter);
            }
            if ( ! src.size() == size())
                throw std::runtime_error("copy_constructor: Copying failed - sizes don’t match up");
        }
    public:

        //--------------------------------------------------
        // iterators
        //--------------------------------------------------
        class SSLL_Const_Iter;
        class SSLL_Iter: public std::iterator<std::forward_iterator_tag, T>
        {
            friend class SSLL_Const_Iter;
        public:
            // inheriting from std::iterator<std::forward_iterator_tag, T>
            // automagically sets up these typedefs...
            typedef T value_type;
            typedef std::ptrdiff_t difference_type;
            typedef T& reference;
            typedef T* pointer;
            typedef std::forward_iterator_tag iterator_category;

            // but not these typedefs...
            typedef SSLL_Iter self_type;
            typedef SSLL_Iter& self_reference;

        private:
            Node* here;

        public:
            explicit SSLL_Iter(Node* start) : here(start) {
                if (start == nullptr)
                    throw std::runtime_error("SSLL_Iter: start cannot be null");
            }
            SSLL_Iter(const SSLL_Iter& src) : here(src.here) {}
            reference operator*() const {
                if (here->is_dummy)
                    throw std::out_of_range("SSLL_Iter: can’t dereference end position");
                return here->item;
            }
            pointer operator->() const {
                return & this->operator*();
            }
            self_reference operator=( const self_type& src ) {
                if (&src == this)
                    return *this;
                here = src.here;
                if (*this != src)
                    throw std::runtime_error("SSLL_Iter: copy assignment failed");
                return *this;
            }
            self_reference operator++() { // preincrement
                if (here->is_dummy)
                    throw std::out_of_range("SSLL_Iter: Can't traverse past the end of the list");
                here = here->next;
                return *this;
            }
            self_type operator++(int) { // postincrement
                self_type t(*this); //save state
                operator++(); //apply increment
                return t; //return state held before increment
            }
            bool operator==(const self_type& rhs) const {
                return rhs.here == here;
            }
            bool operator!=(const self_type& rhs) const {
                return ! operator==(rhs);
            }
        };

        class SSLL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
        {
        public:
            // inheriting from std::iterator<std::forward_iterator_tag, T>
            // automagically sets up these typedefs...
            typedef T value_type;
            typedef std::ptrdiff_t difference_type;
            typedef const T& reference;
            typedef const T* pointer;
            typedef std::forward_iterator_tag iterator_category;

            // but not these typedefs...
            typedef SSLL_Const_Iter self_type;
            typedef SSLL_Const_Iter& self_reference;

        private:
            const Node* here;

        public:
            explicit SSLL_Const_Iter(Node* start) : here(start) {
                if (start == nullptr)
                    throw std::runtime_error("SSLL_Const_Iter: start cannot be null");
            }
            SSLL_Const_Iter(const SSLL_Const_Iter& src) : here(src.here) {}
            SSLL_Const_Iter(const SSLL_Iter& src) : here(src.here) {}

            reference operator*() const {
                if (here->is_dummy)
                    throw std::out_of_range("SSLL_Const_Iter: can't dereference end position");
                return here->item;
            }
            pointer operator->() const {
                return & this->operator*();
            }
            self_reference operator=( const self_type& src ) {
                if (&src == this)
                    return *this;
                here = src.here;
                if (*this != src)
                    throw std::runtime_error("SSLL_Const_Iter: copy assignment failed");
                return *this;
            }
            self_reference operator++() { // preincrement
                if (here->is_dummy)
                    throw std::out_of_range("SSLL_Const_Iter: Can't traverse past the end of the list");
                here = here->next;
                return *this;
            }
            self_type operator++(int) { // postincrement
                self_type t(*this); //save state
                operator++(); //apply increment
                return t; //return state held before increment
            }
            bool operator==(const self_type& rhs) const {
                return rhs.here == here;
            }
            bool operator!=(const self_type& rhs) const {
                return ! operator==(rhs);
            }
        };

        //--------------------------------------------------
        // types
        //--------------------------------------------------
        typedef T value_type;
        typedef SSLL_Iter iterator;
        typedef SSLL_Const_Iter const_iterator;

        iterator begin() { return SSLL_Iter(head->next); }
        iterator end() { return SSLL_Iter(tail); }

        const_iterator begin() const { return SSLL_Const_Iter(head->next); }
        const_iterator end() const { return SSLL_Const_Iter(tail); }

        //--------------------------------------------------
        // operators
        //--------------------------------------------------
        T& operator[](size_t i) {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at position ") + std::to_string(i));
            }
            return node_at(i)->item;
        }

        const T& operator[](size_t i) const {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at position ") + std::to_string(i));
            }
            return node_at(i)->item;
        }

        //--------------------------------------------------
        // Constructors/destructor/assignment operator
        //--------------------------------------------------

        SSLL() {
            init();
        }
        //--------------------------------------------------
        //copy constructor
        //note to self: src must be const in case we want to assign this from a const source
        SSLL(const SSLL& src) {
            init();
            copy_constructor(src);
        }

        //--------------------------------------------------
        //destructor
        ~SSLL() {
            // safely dispose of this SSLL's contents
            clear();
        }

        //--------------------------------------------------
        //copy assignment constructor
        SSLL& operator=(const SSLL& src) {
            if (&src == this) // check for self-assignment
                return *this;   // do nothing
            // safely dispose of this SSLL's contents
            clear();
            // populate this SSLL with copies of the other SSLL's contents
            copy_constructor(src);
            return *this;
        }

        //--------------------------------------------------
        // member functions
        //--------------------------------------------------

        /*
            replaces the existing element at the specified position with the specified element and
            returns the original element.
        */
        T replace(const T& element, size_t position) {
            T old_item;
            if (position >= size()) {
                throw std::out_of_range(std::string("replace: No element at position ") + std::to_string(position));
            } else {
                //we are guaranteed to be at a non-dummy item now because of the above if statement
                Node* iter = node_at(position);
                old_item = iter->item;
                iter->item = element;
            }
            return old_item;
        }

        //--------------------------------------------------
        /*
            adds 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.
        */
        void insert(const T& element, size_t position) {
            if (position > size()) {
                throw std::out_of_range(std::string("insert: Position is outside of the list: ") + std::to_string(position));
            } else if (position == size()) {
                //special O(1) case
                push_back(element);
            } else {
                //node_before_position is guaranteed to point to a valid node because we use a dummy head node
                Node* node_before_position = node_before(position);
                Node* node_at_position = node_before_position->next;
                Node* new_node;
                try {
                    new_node = design_new_node(element, node_at_position);
                } catch (std::bad_alloc& ba) {
                    std::cerr << "insert(): failed to allocate memory for new node"
                        << std::endl;
                    throw std::bad_alloc();
                }
                insert_node_after(node_before_position, new_node);
            }
        }

        /*
            prepends the specified element to the list.
        */
        void push_front(const T& element) {
            insert(element, 0);
        }

        //--------------------------------------------------
        /*
            appends the specified element to the list.
        */
        void push_back(const T& element) {
            Node* new_tail;
            try {
                new_tail = design_new_node(nullptr, true);
            } catch (std::bad_alloc& ba) {
                std::cerr << "push_back(): failed to allocate memory for new tail"
                    << std::endl;
                throw std::bad_alloc();
            }
            insert_node_after(tail, new_tail);
            //transform the current tail node from a dummy to a real node holding
                element
            tail->is_dummy = false;
            tail->item = element;
            tail->next = new_tail;
            tail = tail->next;
        }

        /*
            removes and returns the element at the list's head.
        */
        T pop_front() {
            if (is_empty()) {
                throw std::out_of_range("pop_front: Can't pop: list is empty");
            }
            if (head->next == tail) {
                throw std::runtime_error("pop_front: head->next == tail, but list
                    says it's not empty (corrupt state)");
            }
            return remove_item_after(head);
        }

        //--------------------------------------------------
        /*
            removes and returns the element at the list’s tail.
        */
        T pop_back() {
            if (is_empty()) {
                throw std::out_of_range("pop_back: Can’t pop: list is empty");
            }
            if (head->next == tail) {
                throw std::runtime_error("pop_back: head->next == tail, but list says it’s not empty (corrupt state)");
            }
            //XXX this is O(N), a disadvantage of this architecture
            Node* node_before_last = node_before(size() - 1);
            T item = remove_item_after(node_before_last);
            return item;
        }

        //--------------------------------------------------
        /*
            removes and returns the the element at the specified position,
            shifting the subsequent elements one position to the left.
        */
        T remove(size_t position) {
            T item;
            if (position >= size()) {
                throw std::out_of_range(std::string("remove: No element at position ") + std::to_string(position));
            }
            if (head->next == tail) {
                throw std::runtime_error("remove: head->next == tail, but list says it’s not empty (corrupt state)");
            }
            //using a dummy head node guarantees that there be a node immediately preceeding the specified position
            Node *node_before_position = node_before(position);
            item = remove_item_after(node_before_position);
            return item;
        }

        //--------------------------------------------------
        /*
            returns (without removing from the list) the element at the specified position.
        */
        T item_at(size_t position) const {
            if (position >= size()) {
                throw std::out_of_range(std::string("item_at: No element at position ") + std::to_string(position));
            }
            return operator[](position);
        }

```

```
    //--------------------------------------------------
    /*
        returns true IFF the list contains no elements.
    */
    bool is_empty() const {
        return size() == 0;
    }

    //--------------------------------------------------
    /*
        returns the number of elements in the list.
    */
    size_t size() const {
        if (num_items == 0 && head->next != tail) {
            throw std::runtime_error("size: head->next != tail, but list says
                    it's empty (corrupt state)");
        } else if (num_items > 0 && head->next == tail) {
            throw std::runtime_error("size: head->next == tail, but list says
                    it's not empty (corrupt state)");
        }
        return num_items;
    }

    //--------------------------------------------------
    /*
        removes all elements from the list.
    */
    void clear() {
        while ( ! is_empty()) {
            pop_front();
        }
    }

    //--------------------------------------------------
    /*
        returns true IFF one of the elements of the list matches the specified
            element.
    */
    bool contains(const T& element,
          bool equals(const T& a, const T& b)) const {
        bool element_in_list = false;
        const_iterator fin = end();
        for (const_iterator iter = begin(); iter != fin; ++iter) {
            if (equals(*iter, element)) {
                element_in_list = true;
                break;
            }
        }
        return element_in_list;
    }

    //--------------------------------------------------
```

```
        /*
            If the list is empty, inserts "<empty list>" into the ostream;
            otherwise, inserts, enclosed in square brackets, the list's elements,
            separated by commas, in sequential order.
        */
        std::ostream& print(std::ostream& out) const {
            if (is_empty()) {
                out << "<empty list>";
            } else {
                out << "[";
                const_iterator start = begin();
                const_iterator fin = end();
                for (const_iterator iter = start; iter != fin; ++iter) {
                    if (iter != start)
                        out << ",";
                    out << *iter;
                }
                out << "]";
            }
            return out;
        }
    protected:
        bool validate_internal_integrity() {
            //todo: fill this in
            return true;
        }
    }; //end class SSLL
} // end namespace cop3530
#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) {
```

```
                std::cerr << "procure_free_node(): failed to allocate new node" << std::endl;
                throw std::bad_alloc();
            }
        } else {
            n = remove_node_after(free_list_head, num_free_list_items);
        }
        return n;
    }
    void shrink_pool_if_necessary() {
        if (size() >= 100) {
            size_t old_size = size();
            while (free_list_size() > size() / 2) { //while the pool contains more nodes than half the list size
                Node* n = remove_node_after(free_list_head, num_free_list_items);
                delete n;
            }
        }
    }

    size_t free_list_size() { return num_free_list_items; }
    Node* design_new_node(const T& element, Node* next = nullptr, bool dummy = false, bool force_allocation = false) {
        Node* n = procure_free_node(force_allocation);
        n->is_dummy = dummy;
        n->item = element;
        n->next = next;
        return n;
    }
    Node* design_new_node(Node* next = nullptr, bool dummy = false, bool force_allocation = false) {
        Node* n = procure_free_node(force_allocation);
        n->is_dummy = dummy;
        n->next = next;
        return n;
    }
    void init() {
        num_main_list_items = 0;
        num_free_list_items = 0;
        free_list_head = design_new_node(nullptr, true, true);
        tail = design_new_node(nullptr, true, true);
        head = design_new_node(tail, true, true);
    }
    void copy_constructor(const PSLL& src) {
        //note: this function does *not* copy the free list
        const_iterator fin = src.end();
        for (const_iterator iter = src.begin(); iter != fin; ++iter) {
            push_back(*iter);
        }
        if ( ! src.size() == size())
            throw std::runtime_error("copy_constructor: Copying failed - sizes don't match up");
    }
    Node* remove_node_after(Node* preceeding_node, size_t& list_size_counter) {
        if (preceeding_node->next == tail) {
            throw std::runtime_error("remove_node_after: preceeding_node->next==tail, and we cant remove the tail");
        }
        if (preceeding_node == tail) {
            throw std::runtime_error("remove_node_after: preceeding_node==tail, and we cant remove after the tail");
        }
        if (preceeding_node == free_list_head && free_list_size() == 0) {
            throw std::runtime_error("remove_node_after: attempt detected to remove a node from an empty pool");
        }
        Node* removed_node = preceeding_node->next;
        preceeding_node->next = removed_node->next;
        removed_node->next = nullptr;
        --list_size_counter;
        return removed_node;
    }

    void insert_node_after(Node* existing_node, Node* new_node, size_t& list_size_counter) {
        new_node->next = existing_node->next;
        existing_node->next = new_node;
        ++list_size_counter;
    }

    //returns subsequent node's item and moves that node to the free pool
    T remove_item_after(Node* preceeding_node) {
        Node* removed_node = remove_node_after(preceeding_node, num_main_list_items);
        T item = removed_node->item;
        insert_node_after(free_list_head, removed_node, num_free_list_items);
        shrink_pool_if_necessary();
        return item;
    }

public:
    //--------------------------------------------------
    // iterators
    //--------------------------------------------------
    class PSLL_Const_Iter;
    class PSLL_Iter: public std::iterator<std::forward_iterator_tag, T>
    {
        friend class PSLL_Const_Iter;
    public:
        // inheriting from std::iterator<std::forward_iterator_tag, T>
        // automagically sets up these typedefs...
        typedef T value_type;
        typedef std::ptrdiff_t difference_type;
        typedef T& reference;
        typedef T* pointer;
        typedef std::forward_iterator_tag iterator_category;

        // but not these typedefs...
        typedef PSLL_Iter self_type;
        typedef PSLL_Iter& self_reference;

    private:
        Node* here;

    public:
        explicit PSLL_Iter(Node* start) : here(start) {
            if (start == nullptr)
                throw std::runtime_error("PSLL_Iter: start cannot be null");
        }
        PSLL_Iter(const PSLL_Iter& src) : here(src.here) {}
        reference operator*() const {
            if (here->is_dummy)
                throw std::out_of_range("SSLL_Iter: can't dereference end
                    position");
            return here->item;
        }
        pointer operator->() const {
            return & this->operator*();
        }
        self_reference operator=( const self_type& src ) {
            if (&src == this)
                return *this;
            here = src.here;
            if (*this != src)
                throw std::runtime_error("PSLL_Iter: copy assignment failed");
            return *this;
        }
        self_reference operator++() { // preincrement
            if (here->is_dummy)
                throw std::out_of_range("PSLL_Iter: Can't traverse past the end
                    of the list");
            here = here->next;
            return *this;
        }
        self_type operator++(int) { // postincrement
            self_type t(*this); //save state
            operator++(); //apply increment
            return t; //return state held before increment
        }
        bool operator==(const self_type& rhs) const {
            return rhs.here == here;
        }
        bool operator!=(const self_type& rhs) const {
            return ! operator==(rhs);
        }
    };
```

```
        class PSLL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
        {
        public:
            // inheriting from std::iterator<std::forward_iterator_tag, T>
            // automagically sets up these typedefs...
            typedef T value_type;
            typedef std::ptrdiff_t difference_type;
            typedef const T& reference;
            typedef const T* pointer;
            typedef std::forward_iterator_tag iterator_category;

            // but not these typedefs...
            typedef PSLL_Const_Iter self_type;
            typedef PSLL_Const_Iter& self_reference;

        private:
            const Node* here;

        public:
            explicit PSLL_Const_Iter(Node* start) : here(start) {
                if (start == nullptr)
                    throw std::runtime_error("PSLL_Const_Iter: start cannot be null");
            }
            PSLL_Const_Iter(const PSLL_Const_Iter& src) : here(src.here) {}
            PSLL_Const_Iter(const PSLL_Iter& src) : here(src.here) {}

            reference operator*() const {
                if (here->is_dummy)
                    throw std::out_of_range("SSLL_Iter: can't dereference end position");
                return here->item;
            }
            pointer operator->() const {
                return & this->operator*();
            }
            self_reference operator=( const self_type& src ) {
                if (&src == this)
                    return *this;
                here = src.here;
                if (*this != src)
                    throw std::runtime_error("PSLL_Const_Iter: copy assignment failed");
                return *this;
            }
            self_reference operator++() { // preincrement
                if (here->is_dummy)
                    throw std::out_of_range("PSLL_Const_Iter: Can't traverse past the end of the list");
                here = here->next;
                return *this;
            }
            self_type operator++(int) { // postincrement
                self_type t(*this); //save state
                operator++(); //apply increment
                return t; //return state held before increment
            }
            bool operator==(const self_type& rhs) const {
                return rhs.here == here;
            }
            bool operator!=(const self_type& rhs) const {
                return ! operator==(rhs);
            }
        };

        //---------------------------------------------------
        // types
        //---------------------------------------------------
        /*typedef std::size_t size_t;*/
        typedef T value_type;
        typedef PSLL_Iter iterator;
        typedef PSLL_Const_Iter const_iterator;

        iterator begin() {
            return iterator(head->next);
        }
        iterator end() {
            return iterator(tail);
        }
        /*
            Note to self: the following overloads will fail if not defined as const
        */
        const_iterator begin() const {
            return const_iterator(head->next);
        }
        const_iterator end() const {
            return const_iterator(tail);
        }

        //---------------------------------------------------
        // operators
        //---------------------------------------------------
        T& operator[](size_t i) {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at position ") + std::to_string(i));
            }
            return node_at(i)->item;
        }

        const T& operator[](size_t i) const {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at position ") + std::to_string(i));
            }
            return node_at(i)->item;
        }

        //--------------------------------------------------
        // Constructors/destructor/assignment operator
        //--------------------------------------------------

        PSLL() {
            init();
        }
        //--------------------------------------------------
        //copy constructor
        PSLL(const PSLL& src) {
            init();
            copy_constructor(src);
        }

        //--------------------------------------------------
        //destructor
        ~PSLL() {
            // safely dispose of this PSLL's contents
            clear();
        }

        //--------------------------------------------------
        //copy assignment constructor
        PSLL& operator=(const PSLL& src) {
            if (&src == this) // check for self-assignment
                return *this;   // do nothing
            // safely dispose of this PSLL's contents
            clear();
            // populate this PSLL with copies of the other PSLL's contents
            copy_constructor(src);
            return *this;
        }

        //--------------------------------------------------
        // member functions
        //--------------------------------------------------

        /*
            replaces the existing element at the specified position with the specified element and
            returns the original element.
        */
        T replace(const T& element, size_t position) {
            T old_item;
            if (position >= size()) {
                throw std::out_of_range(std::string("replace: No element at position ") + std::to_string(position));
            } else {
```

```cpp
                //we are guaranteed to be at a non-dummy item now because of the above if statement
                Node* iter = node_at(position);
                old_item = iter->item;
                iter->item = element;
            }
            return old_item;
        }

        //--------------------------------------------------
        /*
            adds 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.
        */
        void insert(const T& element, size_t position) {
            if (position > size()) {
                throw std::out_of_range(std::string("insert: Position is outside of the list: ") + std::to_string(position));
            } else if (position == size()) {
                //special O(1) case
                push_back(element);
            } else {
                //node_before_position is guaranteed to point to a valid node because we use a dummy head node
                Node* node_before_position = node_before(position);
                Node* node_at_position = node_before_position->next;
                Node* new_node;
                try {
                    new_node = design_new_node(element, node_at_position);
                } catch (std::bad_alloc& ba) {
                    std::cerr << "insert(): failed to allocate memory for new node" << std::endl;
                    throw std::bad_alloc();
                }
                insert_node_after(node_before_position, new_node, num_main_list_items);
            }
        }

        //--------------------------------------------------
        //Note to self: use reference here because we receive the original object instance,
        //then copy it into n->item so we have it if the original element goes out of scope
        /*
            prepends the specified element to the list.
        */
        void push_front(const T& element) {
            insert(element, 0);
        }
```

```

    //--------------------------------------------------
    /*
        appends the specified element to the list.
    */
    void push_back(const T& element) {
        Node* new_tail;
        try {
            new_tail = design_new_node(nullptr, true);
        } catch (std::bad_alloc& ba) {
            std::cerr << "push_back(): failed to allocate memory for new tail" << std::endl;
            throw std::bad_alloc();
        }
        insert_node_after(tail, new_tail, num_main_list_items);
        //transform the current tail node from a dummy to a real node holding element
        tail->is_dummy = false;
        tail->item = element;
        tail->next = new_tail;
        tail = tail->next;
    }

    //--------------------------------------------------
    //Note to self: no reference here, so we get our copy of the item, then return a copy
    //of that so the client still has a valid instance if our destructor is called
    /*
        removes and returns the element at the list’s head.
    */
    T pop_front() {
        if (is_empty()) {
            throw std::out_of_range("pop_front: Can’t pop: list is empty");
        }
        if (head->next == tail) {
            throw std::runtime_error("pop_front: head->next == tail, but list says it’s not empty (corrupt state)");
        }
        return remove_item_after(head);
    }

    //--------------------------------------------------
    /*
        removes and returns the element at the list’s tail.
    */
    T pop_back() {
        if (is_empty()) {
            throw std::out_of_range("pop_back: Can’t pop: list is empty");
        }
        if (head->next == tail) {
            throw std::runtime_error("pop_back: head->next == tail, but list says it’s not empty (corrupt state)");
        }
        //XXX this is O(N), a disadvantage of this architecture
        Node* node_before_last = node_before(size() - 1);
        T item = remove_item_after(node_before_last);
        return item;
    }

    //---------------------------------------------------
    /*
        removes and returns the the element at the specified position,
        shifting the subsequent elements one position to the left.
    */
    T remove(size_t position) {
        T item;
        if (position >= size()) {
            throw std::out_of_range(std::string("remove: No element at position ") + std::to_string(position));
        }
        if (head->next == tail) {
            throw std::runtime_error("remove: head->next == tail, but list says it’s not empty (corrupt state)");
        }
        //using a dummy head node guarantees that there be a node immediately preceeding the specified position
        Node *node_before_position = node_before(position);
        item = remove_item_after(node_before_position);
        return item;
    }
    //---------------------------------------------------
    /*
        returns (without removing from the list) the element at the specified position.
    */
    T item_at(size_t position) const {
        if (position >= size()) {
            throw std::out_of_range(std::string("item_at: No element at position ") + std::to_string(position));
        }
        return operator[](position);
    }


    //---------------------------------------------------
    /*
        returns true IFF the list contains no elements.
    */
    bool is_empty() const {
        return size() == 0;
    }

    //--------------------------------------------------
    /*
        returns the number of elements in the list.
    */
    size_t size() const {
        if (num_main_list_items == 0 && head->next != tail) {
            throw std::runtime_error("size: head->next != tail, but list says
                it’s empty (corrupt state)");
        } else if (num_main_list_items > 0 && head->next == tail) {
            throw std::runtime_error("size: head->next == tail, but list says
                it’s not empty (corrupt state)");
        }
        return num_main_list_items;
    }

    //--------------------------------------------------
    /*
        removes all elements from the list.
    */
    void clear() {
        while (size()) {
            pop_front();
        }
    }
    //--------------------------------------------------
    /*
        returns true IFF one of the elements of the list matches the specified
            element.
    */
    bool contains(const T& element,
          bool equals(const T& a, const T& b)) const {
        bool element_in_list = false;
        const_iterator fin = end();
        for (const_iterator iter = begin(); iter != fin; ++iter) {
            if (equals(*iter, element)) {
                element_in_list = true;
                break;
            }
        }
        return element_in_list;
    }

    //--------------------------------------------------
    /*
        If the list is empty, inserts "<empty list>" into the ostream;
        otherwise, inserts, enclosed in square brackets, the list’s elements,
        separated by commas, in sequential order.
    */
    std::ostream& print(std::ostream& out) const {
        if (is_empty()) {
            out << "<empty list>";
        } else {
                out << "[";
                const_iterator start = begin();
                const_iterator fin = end();
                for (const_iterator iter = start; iter != fin; ++iter) {
                    if (iter != start)
                        out << ",";
                    out << *iter;
                }
                out << "]";
            }
            return out;
        }
    protected:
        bool validate_internal_integrity() {
            //todo: fill this in
            return true;
        }
    }; //end class PSLL
} // end namespace cop3530
#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) {
                size_t new_array_size = ceil(array_size * 1.5);
                T* new_item_array = allocate_nodes(new_array_size);
                for (size_t i = 0; i != filled_slots; ++i) {
                    new_item_array[i] = item_array[i];
                }
                delete[] item_array;
                item_array = new_item_array;
                array_size = new_array_size;
                ++embiggen_counter;
            }
        }
        void shrink_if_necessary() {
            /*
                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.
            */
            size_t filled_slots = size();
            if (array_size >= 100 && filled_slots < array_size / 2) {
                size_t new_array_size = ceil(array_size * 0.5);
                T* new_item_array = allocate_nodes(new_array_size);
                for (size_t i = 0; i != filled_slots; ++i) {
                    new_item_array[i] = item_array[i];
                }
                delete[] item_array;
                item_array = new_item_array;
                array_size = new_array_size;
                ++shrink_counter;
            }
        }
        void init(size_t num_nodes_to_preallocate) {
            array_size = num_nodes_to_preallocate;
            num_items = 0;
            item_array = allocate_nodes(array_size);
        }
        void copy_constructor(const SDAL& src) {
            const_iterator fin = src.end();
            for (const_iterator iter = src.begin(); iter != fin; ++iter) {
                push_back(*iter);
            }
            if ( ! src.size() == size())
                throw std::runtime_error("copy_constructor: Copying failed - sizes don't match up");
        }
    public:

        //--------------------------------------------------
        // iterators
        //--------------------------------------------------
        class SDAL_Const_Iter;
        class SDAL_Iter: public std::iterator<std::forward_iterator_tag, T>
        {
            friend class SDAL_Const_Iter;
        public:
            // inheriting from std::iterator<std::forward_iterator_tag, T>
            // automagically sets up these typedefs...
            //todo: figure out why we cant comment these out, which we should be able to if they were
            //defined when inheriting
            typedef T value_type;
            typedef std::ptrdiff_t difference_type;
            typedef T& reference;
            typedef T* pointer;
            typedef std::forward_iterator_tag iterator_category;

            // but not these typedefs...
            typedef SDAL_Iter self_type;
            typedef SDAL_Iter& self_reference;

        private:
            T* iter;
            T* end_iter;

        public:
            explicit SDAL_Iter(T* item_array, T* end_ptr): iter(item_array), end_iter(end_ptr) {
                if (item_array == nullptr)
                    throw std::runtime_error("SDAL_Iter: item_array cannot be null");
                if (end_ptr == nullptr)
                    throw std::runtime_error("SDAL_Iter: end_ptr cannot be null");
                if (item_array > end_ptr)
                    throw std::runtime_error("SDAL_Iter: item_array pointer cannot be past end_ptr");
            }
            SDAL_Iter(const SDAL_Iter& src): iter(src.iter), end_iter(src.end_iter) {}
            reference operator*() const {
                if (iter == end_iter)
                    throw std::out_of_range("SDAL_Iter: can’t dereference end position");
                return *iter;
            }
            pointer operator->() const {
                return & this->operator*();
            }
            self_reference operator=( const self_type& src ) {
                if (&src == this)
                    return *this;
                iter = src.iter;
                end_iter = src.end_iter;
                if (*this != src)
                    throw std::runtime_error("SDAL_Iter: copy assignment failed");
                return *this;
            }
            self_reference operator++() { // preincrement
                if (iter == end_iter)
                    throw std::out_of_range("SDAL_Iter: Can’t traverse past the end of the list");
                ++iter;
                return *this;
            }
            self_type operator++(int) { // postincrement
                self_type t(*this); //save state
                operator++(); //apply increment
                return t; //return state held before increment
            }
            bool operator==(const self_type& rhs) const {
                return rhs.iter == iter && rhs.end_iter == end_iter;
            }
            bool operator!=(const self_type& rhs) const {
                return ! operator==(rhs);
            }
        };

        class SDAL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
        {
        public:
            // inheriting from std::iterator<std::forward_iterator_tag, T>
            // automagically sets up these typedefs...
            typedef T value_type;
            typedef std::ptrdiff_t difference_type;
            typedef const T& reference;
            typedef const T* pointer;
            typedef std::forward_iterator_tag iterator_category;

            // but not these typedefs...
            typedef SDAL_Const_Iter self_type;
            typedef SDAL_Const_Iter& self_reference;
        private:
            const T* iter;
            const T* end_iter;
        public:
            explicit SDAL_Const_Iter(T* item_array, T* end_ptr): iter(item_array), end_iter(end_ptr) {
                if (item_array == nullptr)
                    throw std::runtime_error("SDAL_Const_Iter: item_array cannot be null");
                if (end_ptr == nullptr)
                    throw std::runtime_error("SDAL_Const_Iter: end_ptr cannot be null");
                if (item_array > end_ptr)
                    throw std::runtime_error("SDAL_Const_Iter: item_array pointer cannot be past end_ptr");
            }
```

```
        SDAL_Const_Iter(const SDAL_Const_Iter& src): iter(src.iter), end_iter(src.end_iter) {}
        SDAL_Const_Iter(const SDAL_Iter& src): iter(src.iter), end_iter(src.end_iter) {}
        reference operator*() const {
            if (iter == end_iter)
                throw std::out_of_range("SDAL_Const_Iter: can’t dereference end position");
            return *iter;
        }
        pointer operator->() const {
            return & this->operator*();
        }
        self_reference operator=( const self_type& src ) {
            if (&src == this)
                return *this;
            iter = src.iter;
            end_iter = src.end_iter;
            if (*this != src)
                throw std::runtime_error("SDAL_Const_Iter: copy assignment failed");
            return *this;
        }
        self_reference operator++() { // preincrement
            if (iter == end_iter)
                throw std::out_of_range("SDAL_Const_Iter: Can’t traverse past the end of the list");
            ++iter;
            return *this;
        }
        self_type operator++(int) { // postincrement
            self_type t(*this); //save state
            operator++(); //apply increment
            return t; //return state held before increment
        }
        bool operator==(const self_type& rhs) const {
            return rhs.iter == iter && rhs.end_iter == end_iter;
        }
        bool operator!=(const self_type& rhs) const {
            return ! operator==(rhs);
        }
    };

    //--------------------------------------------------
    // types
    //--------------------------------------------------
    typedef T value_type;
    typedef SDAL_Iter iterator;
    typedef SDAL_Const_Iter const_iterator;

    iterator begin() { return SDAL_Iter(item_array, item_array + num_items); }
```

```
        iterator end() { return SDAL_Iter(item_array + num_items, item_array + num_items); }

        const_iterator begin() const { return SDAL_Const_Iter(item_array, item_array + num_items); }
        const_iterator end() const { return SDAL_Const_Iter(item_array + num_items, item_array + num_items); }

        //--------------------------------------------------
        // operators
        //--------------------------------------------------
        T& operator[](size_t i) {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at position ") + std::to_string(i));
            }
            return item_array[i];
        }

        const T& operator[](size_t i) const {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at position ") + std::to_string(i));
            }
            return item_array[i];
        }

        //--------------------------------------------------
        // Constructors/destructor/assignment operator
        //--------------------------------------------------

        SDAL(size_t num_nodes_to_preallocate = 50) {
            init(num_nodes_to_preallocate);
        }

        //--------------------------------------------------
        //copy constructor
        SDAL(const SDAL& src): SDAL(src.array_size) {
            init(src.array_size);
            copy_constructor(src);
        }

        //--------------------------------------------------
        //destructor
        ~SDAL() {
            // safely dispose of this SDAL's contents
            delete[] item_array;
        }

        //--------------------------------------------------
        //copy assignment constructor
        SDAL& operator=(const SDAL& src) {
            if (&src == this) // check for self-assignment
                return *this;   // do nothing
            delete[] item_array;
            init(src.array_size);
            copy_constructor(src);
            return *this;
        }

        //--------------------------------------------------
        // member functions
        //--------------------------------------------------

        /*
            replaces the existing element at the specified position with the specified element and
            returns the original element.
        */
        T replace(const T& element, size_t position) {
            T old_item;
            if (position >= size()) {
                throw std::out_of_range(std::string("replace: No element at position ") + std::to_string(position));
            } else {
                old_item = item_array[position];
                item_array[position] = element;
            }
            return old_item;
        }

        //--------------------------------------------------
        /*
            adds 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.
        */
        void insert(const T& element, size_t position) {
            if (position > size()) {
                throw std::out_of_range(std::string("insert: Position is outside of the list: ") + std::to_string(position));
            } else {
                embiggen_if_necessary();
                //shift remaining items right
                for (size_t i = size(); i != position; --i) {
                    item_array[i] = item_array[i - 1];
                }
                item_array[position] = element;
                ++num_items;
            }
        }

        //--------------------------------------------------
        //Note to self: use reference here because we receive the original object instance,
        //then copy it into n->item so we have it if the original element goes out of scope
        /*
            prepends the specified element to the list.
        */
        void push_front(const T& element) {
            insert(element, 0);
        }

        //--------------------------------------------------
        /*
            appends the specified element to the list.
        */
        void push_back(const T& element) {
            insert(element, size());
        }


        //--------------------------------------------------
        //Note to self: no reference here, so we get our copy of the item, then return a copy
        //of that so the client still has a valid instance if our destructor is called
        /*
            removes and returns the element at the list's head.
        */
        T pop_front() {
            if (is_empty()) {
                throw std::out_of_range("pop_front: Can't pop: list is empty");
            }
            return remove(0);
        }

        //--------------------------------------------------
        /*
            removes and returns the element at the list's tail.
        */
        T pop_back() {
            if (is_empty()) {
                throw std::out_of_range("pop_back: Can't pop: list is empty");
            }
            return remove(size() - 1);
        }

        //--------------------------------------------------
        /*
            removes and returns the the element at the specified position,
            shifting the subsequent elements one position to the left.
        */
        T remove(size_t position) {
```

```
        T item;
        if (position >= size()) {
            throw std::out_of_range(std::string("remove: No element at position ") + std::to_string(position));
        } else {
            item = item_array[position];
            //shift remaining items left
            for (size_t i = position + 1; i != size(); ++i) {
                item_array[i - 1] = item_array[i];
            }
            --num_items;
            shrink_if_necessary();
        }
        return item;
    }

    //-------------------------------------------------
    /*
        returns (without removing from the list) the element at the specified position.
    */
    T item_at(size_t position) const {
        if (position >= size()) {
            throw std::out_of_range(std::string("item_at: No element at position ") + std::to_string(position));
        }
        return operator[](position);
    }

    //-------------------------------------------------
    /*
        returns true IFF the list contains no elements.
    */
    bool is_empty() const {
        return size() == 0;
    }

    //-------------------------------------------------
    /*
        returns the number of elements in the list.
    */
    size_t size() const {
        return num_items;
    }

    //-------------------------------------------------
    /*
        removes all elements from the list.
    */
    void clear() {
        //no reason to do memory deallocation here, just overwrite the old items later and save
```

```
            //deallocation for the deconstructor
            num_items = 0;
        }

        //--------------------------------------------------
        /*
            returns true IFF one of the elements of the list matches the specified element.
        */
        bool contains(const T& element,
              bool equals(const T& a, const T& b)) const {
            bool element_in_list = false;
            const_iterator fin = end();
            for (const_iterator iter = begin(); iter != fin; ++iter) {
                if (equals(*iter, element)) {
                    element_in_list = true;
                    break;
                }
            }
            return element_in_list;
        }

        //--------------------------------------------------
        /*
            If the list is empty, inserts "<empty list>" into the ostream;
            otherwise, inserts, enclosed in square brackets, the list's elements,
            separated by commas, in sequential order.
        */
        std::ostream& print(std::ostream& out) const {
            if (is_empty()) {
                out << "<empty list>";
            } else {
                out << "[";
                const_iterator start = begin();
                const_iterator fin = end();
                for (const_iterator iter = start; iter != fin; ++iter) {
                    if (iter != start)
                        out << ",";
                    out << *iter;
                }
                out << "]";
            }
            return out;
        }
    protected:
        bool validate_internal_integrity() {
            //todo: fill this in
            return true;
        }
    };
} // end namespace cop3530

```

```
#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

```cpp
#ifndef _CDAL_H_
#define _CDAL_H_

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

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

namespace cop3530 {
    template <class T>
    class CDAL {
    private:
        struct Node {
            //Node is an element in the linked list and contains an array of items
            T* item_array;
            Node* next;
            bool is_dummy;
        };
        struct ItemLoc {
            //ItemLoc describes the position of an item, including its linked list node and position within the array held by that node
            Node* node;
            size_t array_index;
            T& item_ref;
        };
        size_t num_items;
        size_t num_available_nodes; //excludes head/tail nodes
        size_t embiggen_counter = 0;
        size_t shrink_counter = 0;
        Node* head;
        Node* tail;
        static const size_t array_size = 50; //length of each chained array
        Node* node_at(size_t position) const {
            Node* n = head->next;
            for (size_t i = 0; i != position; ++i, n = n->next);
            return n;
        }
        Node* node_before(size_t position) const {
            if (position == 0)
                return head;
            else
                return node_at(position - 1);
        }

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

        Node* design_new_node(Node* next = nullptr, bool dummy = false) const {
            Node* n;
            try {
                n = new Node();
            } catch (std::bad_alloc& ba) {
                std::cerr << "design_new_node(): failed to allocate memory for new node" << std::endl;
                throw std::bad_alloc();
            }
            n->is_dummy = dummy;
            try {
                n->item_array = new T[array_size];
            } catch (std::bad_alloc& ba) {
                std::cerr << "design_new_node(): failed to allocate memory for item array" << std::endl;
                throw std::bad_alloc();
            }
            n->next = next;
            return n;
        }

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

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

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

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

        void embiggen_if_necessary() {
            //embiggen is a perfectly cromulent word
            /*
                If each array slot in every link is filled and we want to add a new item, allocate and append a new link
            */
            if (size() == num_available_nodes * array_size) {
                //transform tail into a regular node and append a new tail
                Node* n = tail;
                n->is_dummy = false;
                tail = n->next = design_new_node(nullptr, true);
                ++num_available_nodes;
                ++embiggen_counter;
            }
        }

        void shrink_if_necessary() {
            /*
                Because we don’t want the list to waste too much memory, whenever the more than half of the arrays
                are unused (they would all be at the end of the chain), deallocate half the unused arrays.
            */
            size_t used = num_used_nodes();
            size_t num_unused_nodes = num_available_nodes - used;
            if (num_unused_nodes > used) {
                size_t nodes_to_keep = used + ceil(num_unused_nodes * 0.5);
                Node* last_node = node_before(nodes_to_keep);
                while (last_node->next != tail) {
                    drop_node_after(last_node);
                }
                ++shrink_counter;
            }
        }
        void copy_constructor(const CDAL& src) {
            const_iterator fin = src.end();
            for (const_iterator iter = src.begin(); iter != fin; ++iter) {
                push_back(*iter);
            }
            if ( ! src.size() == size())
                throw std::runtime_error("copy_constructor: Copying failed - sizes don’t match up");
        }

    public:
        //--------------------------------------------------
        // iterators
        //--------------------------------------------------
        class CDAL_Const_Iter;
        class CDAL_Iter: public std::iterator<std::forward_iterator_tag, T> {
            friend class CDAL_Const_Iter;
        private:
            Node* curr_node;
            size_t curr_array_index;
            Node* fin_node;
            size_t fin_array_index;
        public:
            typedef std::ptrdiff_t difference_type;
            typedef T& reference;
            typedef T* pointer;
            typedef std::forward_iterator_tag iterator_category;
            typedef T value_type;
            typedef CDAL_Iter self_type;
            typedef CDAL_Iter& self_reference;

            //need copy constructor/assigner to make this a first class ADT (doesn't hold pointers that need freeing)
            CDAL_Iter(ItemLoc const& here, ItemLoc const& fin):
                curr_node(here.node),
                curr_array_index(here.array_index),
                fin_node(fin.node),
                fin_array_index(fin.array_index)
            {}
            CDAL_Iter(const self_type& src):
                curr_node(src.curr_node),
                curr_array_index(src.curr_array_index),
                fin_node(src.fin_node),
                fin_array_index(src.fin_array_index)
            {}
            self_reference operator=(const self_type& rhs) {
                //copy assigner
                if (&rhs == this) return *this;
                curr_node = rhs.curr_node;
                curr_array_index = rhs.curr_array_index;
                fin_node = rhs.fin_node;
                fin_array_index = rhs.fin_array_index;

                if (*this != rhs)
                    throw std::runtime_error("CDAL_Iter: copy assignment failed");
                return *this;
            }
            self_reference operator++() { // preincrement
                if (curr_node == fin_node && curr_array_index == fin_array_index)
                    throw std::out_of_range("CDAL_Iter: Can't traverse past the end of the list");
                curr_array_index = (curr_array_index + 1) % array_size;
                if (curr_array_index == 0) curr_node = curr_node->next;
                return *this;
            }
            self_type operator++(int) { // postincrement
                self_type t(*this); //save state
                operator++(); //apply increment
                return t; //return state held before increment
            }
            reference operator*() const {
                if (curr_node == fin_node && curr_array_index == fin_array_index)
                    throw std::out_of_range("SSLL_Iter: can't dereference end
                        position");
                return curr_node->item_array[curr_array_index];
            }
            pointer operator->() const {
                return & this->operator*();
            }
            bool operator==(const self_type& rhs) const {
                return rhs.curr_node == curr_node
                       && rhs.curr_array_index == curr_array_index;
            }
            bool operator!=(const self_type& rhs) const {
                return ! operator==(rhs);
            }
        };

        class CDAL_Const_Iter: public std::iterator<std::forward_iterator_tag, T> {
        private:
            Node* curr_node;
            size_t curr_array_index;
            Node* fin_node;
            size_t fin_array_index;
        public:
            typedef std::ptrdiff_t difference_type;
            typedef const T& reference;
            typedef const T* pointer;
            typedef std::forward_iterator_tag iterator_category;
            typedef T value_type;
            typedef CDAL_Const_Iter self_type;
            typedef CDAL_Const_Iter& self_reference;

            //need copy constructor/assigner to make this a first class ADT (doesn't
                hold pointers that need freeing)
            CDAL_Const_Iter(ItemLoc const& here, ItemLoc const& fin):
                curr_node(here.node),
                curr_array_index(here.array_index),
                fin_node(fin.node),
                fin_array_index(fin.array_index)
            {}
            CDAL_Const_Iter(const self_type& src):
                curr_node(src.curr_node),
                curr_array_index(src.curr_array_index),
                fin_node(src.fin_node),
                fin_array_index(src.fin_array_index)
            {}
            CDAL_Const_Iter(const CDAL_Iter& src):
                curr_node(src.curr_node),
                curr_array_index(src.curr_array_index),
                fin_node(src.fin_node),
                fin_array_index(src.fin_array_index)
            {}
            self_reference operator=(const self_type& rhs) {
                //copy assigner
                if (&rhs == this) return *this;
                curr_node = rhs.curr_node;
                curr_array_index = rhs.curr_array_index;
                fin_node = rhs.fin_node;
                fin_array_index = rhs.fin_array_index;

                if (*this != rhs)
                    throw std::runtime_error("CDAL_Const_Iter: copy assignment failed");
                return *this;
            }
            self_reference operator++() { // preincrement
                if (curr_node == fin_node && curr_array_index == fin_array_index)
                    throw std::out_of_range("CDAL_Const_Iter: Can't traverse past the end of the list");
                curr_array_index = (curr_array_index + 1) % array_size;
                if (curr_array_index == 0) curr_node = curr_node->next;
                return *this;
            }
            self_type operator++(int) { // postincrement
                self_type t(*this); //save state
                operator++(); //apply increment
                return t; //return state held before increment
            }
            reference operator*() const {
                if (curr_node == fin_node && curr_array_index == fin_array_index)
                    throw std::out_of_range("SSLL_Iter: can't dereference end position");
                return curr_node->item_array[curr_array_index];
            }
            pointer operator->() const {
                return & this->operator*();
            }
            bool operator==(const self_type& rhs) const {
                return rhs.curr_node == curr_node
                       && rhs.curr_array_index == curr_array_index;
            }
            bool operator!=(const self_type& rhs) const {
                return ! operator==(rhs);
            }
        };

        //--------------------------------------------------
        // types
        //--------------------------------------------------
        typedef CDAL_Iter iterator;
        typedef CDAL_Const_Iter const_iterator;
        typedef T value_type;
        //todo: might need to add size_t here and other iterators if they were excluded or commented out

        iterator begin() {
            ItemLoc start_loc = loc_from_pos(0);
            ItemLoc end_loc = loc_from_pos(size());
            return iterator(start_loc, end_loc);
        }

        iterator end() {
            ItemLoc end_loc = loc_from_pos(size());
            return iterator(end_loc, end_loc);
        }

        const_iterator begin() const {
            ItemLoc start_loc = loc_from_pos(0);
            ItemLoc end_loc = loc_from_pos(size());
            return const_iterator(start_loc, end_loc);
        }

        const_iterator end() const {
            ItemLoc end_loc = loc_from_pos(size());
            return const_iterator(end_loc, end_loc);
        }

        T& operator[](size_t i) {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at position ") + std::to_string(i));
            }
            return loc_from_pos(i).item_ref;
        }

        const T& operator[](size_t i) const {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at position ") + std::to_string(i));
            }
            return loc_from_pos(i).item_ref;
        }

        //--------------------------------------------------
        // Constructors/destructor/assignment operator
        //--------------------------------------------------

        CDAL() {
            init();
```

```
            embiggen_if_necessary();
        }
        //--------------------------------------------------
        //copy constructor
        CDAL(const CDAL& src) {
            init();
            copy_constructor(src);
        }

        //--------------------------------------------------
        //destructor
        ~CDAL() {
            // safely dispose of this CDAL’s contents
            clear();
        }

        //--------------------------------------------------
        //copy assignment constructor
        CDAL& operator=(const CDAL& src) {
            if (&src == this) // check for self-assignment
                return *this;   // do nothing
            // safely dispose of this CDAL’s contents
            // populate this CDAL with copies of the other CDAL’s contents
            clear();
            init();
            copy_constructor(src);
            return *this;
        }

        //--------------------------------------------------
        // member functions
        //--------------------------------------------------

        /*
            replaces the existing element at the specified position with the specified element and
            returns the original element.
        */
        T replace(const T& element, size_t position) {
            T item = element;
            if (position >= size()) {
                throw std::out_of_range(std::string("replace: No element at position ") + std::to_string(position));
            } else {
                ItemLoc loc = loc_from_pos(position);
                std::swap(loc.item_ref, item);
            }
            return item;
        }

        //--------------------------------------------------
        /*
            adds 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.
        */
        void insert(const T& element, size_t position) {
            if (position > size()) {
                throw std::out_of_range(std::string("insert: Position is outside of the list: ") + std::to_string(position));
            } else {
                embiggen_if_necessary();
                ItemLoc loc = loc_from_pos(position);
                //shift remaining items to the right
                T item_to_insert = element;
                Node* n = loc.node;
                for (size_t i = position; i <= num_items; ++i) {
                    size_t array_index = i % array_size;
                    if ( i != position && array_index == 0 ) {
                        n = n->next;
                    }
                    std::swap(item_to_insert, n->item_array[array_index]);
                }
                ++num_items;
            }
        }

        //--------------------------------------------------
        //Note to self: use reference here because we receive the original object instance,
        //then copy it into n->item so we have it if the original element goes out of scope
        /*
            prepends the specified element to the list.
        */
        void push_front(const T& element) {
            insert(element, 0);
        }

        //--------------------------------------------------
        /*
            appends the specified element to the list.
        */
        void push_back(const T& element) {
            insert(element, size());
        }

        //--------------------------------------------------
        //Note to self: no reference here, so we get our copy of the item, then return a copy
        //of that so the client still has a valid instance if our destructor is called
        /*
```

```
        removes and returns the element at the list’s head.
    */
    T pop_front() {
        if (is_empty()) {
            throw std::out_of_range("pop_front: Can’t pop: list is empty");
        }
        return remove(0);
    }

    //--------------------------------------------------
    /*
        removes and returns the element at the list’s tail.
    */
    T pop_back() {
        if (is_empty()) {
            throw std::out_of_range("pop_back: Can’t pop: list is empty");
        }
        return remove(size() - 1);
    }

    //--------------------------------------------------
    /*
        removes and returns the the element at the specified position,
        shifting the subsequent elements one position to the left.
    */
    T remove(size_t position) {
        T old_item;
        if (position >= size()) {
            throw std::out_of_range(std::string("remove: No element at position ") + std::to_string(position));
        } else {
            ItemLoc loc = loc_from_pos(position);
            //shift remaining items to the left
            Node* n = loc.node;
            old_item = loc.item_ref;
            for (size_t i = position; i != num_items; ++i) {
                size_t curr_array_index = i % array_size;
                size_t next_array_index = (i + 1) % array_size;
                T& curr_item = n->item_array[curr_array_index];
                if ( next_array_index == 0 ) {
                    n = n->next;
                }
                T& next_item = n->item_array[next_array_index];
                std::swap(curr_item, next_item);
            }
            --num_items;
            shrink_if_necessary();
        }
        return old_item;
    }

    //--------------------------------------------------
    /*
        returns (without removing from the list) the element at the specified position.
    */
    T item_at(size_t position) const {
        if (position >= size()) {
            throw std::out_of_range(std::string("item_at: No element at position ") + std::to_string(position));
        }
        return operator[](position);
    }

    //---------------------------------------------------
    /*
        returns true IFF the list contains no elements.
    */
    bool is_empty() const {
        return size() == 0;
    }

    //---------------------------------------------------
    /*
        returns the number of elements in the list.
    */
    size_t size() const {
        return num_items;
    }

    //---------------------------------------------------
    /*
        removes all elements from the list.
    */
    void clear() {
        while (head->next != tail) {
            drop_node_after(head);
        }
        num_items = 0;
    }
    //---------------------------------------------------
    /*
        returns true IFF one of the elements of the list matches the specified element.
    */
    bool contains(const T& element,
          bool equals(const T& a, const T& b)) const {
        bool element_in_list = false;
        const_iterator fin = end();
        for (const_iterator iter = begin(); iter != fin; ++iter) {
            if (equals(*iter, element)) {
                element_in_list = true;
                break;
            }
            }
            return element_in_list;
        }

        //--------------------------------------------------
        /*
            If the list is empty, inserts "<empty list>" into the ostream;
            otherwise, inserts, enclosed in square brackets, the list's elements,
            separated by commas, in sequential order.
        */
        std::ostream& print(std::ostream& out) const {
            if (is_empty()) {
                out << "<empty list>";
            } else {
                out << "[";
                const_iterator start = begin();
                const_iterator fin = end();
                for (const_iterator iter = start; iter != fin; ++iter) {
                    if (iter != start)
                        out << ",";
                    out << *iter;
                }
                out << "]";
            }
            return out;
        }
    }; //end class CDAL
} // end namespace cop3530
#endif // _CDAL_H_
```