Paul Nickerson
COP3530
1087
11/25/2014
Project 1, Parts 1-3

I hereby affirm that the following work is my own and that the Honor Code was neither bent nor broken:

_____

Part 1 Learning Experiences

Part 1 was the perhaps the most challenging aspect of this project since I was very new to C++, and, in fact, my first exposure to modern C++ was through the C++ Primer I read before the semester. I was still getting used to how references worked and the various ways which modern C++ improves the clunky and error-prone C syntax.

Of the four list types, CDAL was, unsurprisingly given its complexity, the most difficult list type to implement. However, I believe it is a very useful list that I will consider using in future projects, since it offers fast random access benefits of a simple array while not requiring contiguous memory, which is useful since I frequently encounter large data sets in my research which may not fit in memory contiguously.

It was during part 1 that I developed a fuzzer which runs random operations against each list type and checks consistency among each (see SSLL Testing Strategies for a description of the fuzzer). This tool proved *extremely* valuable in uncovering bugs. I used it extensively while implementing parts 2-3 to catch bugs early on. My workflow typically looked like 1) write some code, 2) compile it, 3) (optionally) add/update a fuzzer operation to ensure the code path gets hit, 4) run the fuzzer for a while, 5) fix/refactor the code from (1), 6) repeat.

While part 1 was challenging and time consuming, as a learning experience the process was very valuable. Since in the course of my research I do a lot of C++ development, I now incorporate lists into my work quite often. Previously I had never used a linked list for anything and the extent of my list toolset consisted of simple arrays. I now have a much better grasp on the underlying list data structures and how to go about choosing one which meets my needs.

Part 2 Learning Experiences

Part 2 was much easier than part 1 since I had the list classes in place and just needed to add/update things. After updating all the relevant int types to size_t, I have now developed the habit to use size_t almost exclusively when I need a positional unsigned integer, since it alleviates the extra mental overhead of keeping track of overflows. Size_t is almost always big enough to hold positional integer values that I need, and it has the added benefit of preventing hacky code practices like using -1 as a special case value.

I really enjoyed implementing the iterators. I now use iterators all the time and am trying to develop some functional programing habits. When working with large data sets, it is very useful to have a custom iterator class which processes each entry incrementally, rather than fitting the whole thing in memory. For example, in my research one issue I'm facing is external sorting; I have several large files stored on disk in order, and I need to merge them. I solved that by treating each input file as an iterator and using a priority queue to externally merge their contents into a new, sorted iterator, whose value is processed and passed to yet another iterator, etc.

Implementing the iterators presented a great opportunity to refactor a ton of code. Since traversing the list is a common pattern – for example printing each item, copying a list, checking for the existence of an arbitrary item in the list – I went through and converted as many disparate traversal code paths as I could into methods that relied on iterators. This meant lowered complexity and fewer bugs. In fact, it allowed certain methods, such as print() and contains(), to be implemented using the same exact code among the different lists.

Part 3 Learning Experiences

While part 3 wasn't particularly difficult, generating the informal documentation was a huge, time-consuming pain in the neck. I hate writing documentation, so try to write self-documenting code using things like longer, verbose variable/method names, so that reading the code is nearly as natural as reading actual documentation.

That being said, writing the informal documentation for each method was very helpful. While doing so, I found numerous opportunities to refactor code and add thorough exception handling, as well as to scrutinize assumptions I made when writing the methods. So the process of writing the documentation was also a practice in effective code-cleaning. After the documentation was written, knowing what to target with CATCH test cases was trivial.

From the informal documentation, I wrote several sets of CATCH testcases, which collectively should provide a very high degree of code coverage. As described in the fuzzer write-up (see SSLL Testing Strategies), developing the lists alongside testing with the fuzzer led to lists which all behave similarly despite their storage differences. As such it was only necessary to write a single testing suite to target each list type. Prior to zipping the deliverables, my directory structure contained symbolic links which effectively duplicated the testcase CPP files to the various list directories. Each of the testcase files –

random_access.cpp, contains.cpp, iterators.cpp, replace.cpp, insert.cpp, remove.cpp, and copying.cpp – contain a variety of testcases related to the category referred to by the file name. All public members are covered by the testcases. I found that the key to effective testing is to write testcases which treat the lists as pure ADTs (that is, without making any assumptions regarding their implementation behavior aside from attempting to hit generic code paths such as growing/shrinking the list), while the lists themselves rely heavily upon exception handling to validate their internal state. Incidentally I have found that treating classes I am writing as pure ADTs, with the client knowing zilch about the underlying implementation, naturally leads to much cleaner and maintainable code.

The subscript operator introduced another opportunity to refactor. For example, the item_at method is functionally equivalent to passing the subscript operator through the copy constructor. Also, the replace just method does a std::swap of the input element and the subscript operator. Treating these two methods as such allowed me to use the same code to implement them among each of the four list types.

1. Does the program compile without errors?
    1. Yes
2. Does the program compile without warnings?
    1. Yes
3. Does the program run without crashing?
    1. Yes
4. Describe how you tested the program.
    1. Using a list fuzzer described in SSLL Testing Strategies, by manually writing unit tests with the CATCH framework, and by implementing extensive exception handling within the list internals
5. Describe the ways in which the program does *not* meet assignment's specifications.
    1. None
6. Describe all known and suspected bugs.
    1. None
7. Does the program run correctly?
    1. Yes, all output is as expected

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

SSLL Testing Strategy

Paul Nickerson

Fuzz-testing

For the testing requirements of parts 1 and 2, I wrote a list fuzzer that applies pseudo-random operations (starting from a seed value so fuzz runs are reproducable) to each of the four list types, then check to ensure consistency between the lists. An operation is defined as invoking some public list member with a random input. Ideally, the various fuzzing operations should theoretically combine to maximize code coverage. Consistent behavior is defined using the following criteria: after an operation is applied to each list,

- Each list must return an equivalent integer value (analogous to an executable's return code) from the operation. Typically this is the return value of size() called after the method is invoked, but it could also be, for example, a boolean-to-integer cast of the return value from calling contains() or is_empty()
- Each list must report equivalent list contents as determined by the print() function
- If an exception is thrown, the what() method called on each exception must return the same value.
- If the operation involves returning an item, such as calling remove() or item_at(), each operation must return the same item as determined by that item's operator==() member

Operations are generally only a few lines long:

```
size_t i = rand_int(l.size());
l[i] = rv();
ret_item = l.item_at(i);
return l.size();
```

In practice this approach is extremely effective; operations are combined in ways that may be unintuitive for a person trying to come up with testing strategies. It is important, however, to target potentially problematic code paths, such as the growing/shrinking code used by some of the lists. Whenever I wanted to be sure to test these code paths, I would grow and shrink the list acutely:

```
for (int i = 0; i < 1000; ++i) {
    l.push_front(rv());
    l.push_back(rv());
    int sz = l.size();
    int rand_slot = rand_int(sz);
```

```
    l.insert(rv(), rand_slot);
}
for (int i = 0; i < 800; ++i) {
    ret_item = ret_item + l.pop_front();
    ret_item = ret_item + l.pop_back();
    int sz = l.size();
    int rand_slot = rand_int(sz);
    ret_item = ret_item + l.remove(rand_slot);
}
```

Fuzzer architecture

Because usage of polymorphism in the list classes was explicitly prohibited, I wrote a class template, ListFuzzer, which takes a list type as its template parameter. Fuzzer operations are defined as a list of lambda functions and are stored in any of the implemented lists. Since, for example, ListFuzzer<SSLL<...» and ListFuzzer<CDAL<...» are different types and cannot be directly compared, the operation invoker returns an OpResult struct instance, which describes the operation, the list name in std::string format, and the results of performing the operation on that list. As such, there are four OpResults generated during each fuzzing testcase - one for each list type. These are passed via initializer list to the result validation function, which verifies consistency. After each round, the source code of the operation is read from the source code file (determined with the ___LINE___ and ___FILE___ preprocessor macros) and displayed on the console along with the results from running the operation.

The fuzzer increments a counter after each fuzzing round. Whenever inconsistent results are returned, the fuzzer throws an exception. While debugging, this behavior is extremely handy; since all fuzzing behavior stems from a single seed specified at the beginning, the user can set a conditional breakpoint to break when the operation counter equals the value it was when inconsistent state was detected, but before the operation that caused the inconsistent state to occur actually gets called. The user can then step into the suspect code and find the culprit.

Because the fuzzer expects strict consistency of list behavior, even among the text of exceptions that are thrown, using the fuzzer as a supplement to development forced me to use similar patterns between the lists. Several methods, such as the subscript operator, the print function, and the copy constructor, have converged on versions which use exactly the same code in each of the lists. This reduces the number of possible things that can go wrong.

Results

Using the fuzzer regularly during list development uncovered several subtle bugs which likely would not have been found during simple tests. According to Murphy's Law the bugs would therefore have been encountered only in production when the misbehavior of the list would lead to nuclear missiles being launched at orphanages. At first, bugs would be found right away, within the first hundred-or-so operations. As I fixed bugs and refactored list code, bugs started appearing much less frequently. At their current state, the lists appear

to be rock-solid and display consistent behavior even after hundreds of millions of fuzzer operations.

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

PSLL Testing Strategy

Paul Nickerson

Fuzz-testing

To test the Pool-using Singly-Linked List in parts 1 and 2, I used the fuzzer described in the SSLL Testing Strategy. Particular attention was paid to writing fuzzer operations which would insert and remove items frequently in an effort to corrupt the free list.

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

SDAL Testing Strategy

Paul Nickerson

Fuzz-testing

To test the Simple Dynamic Array-based List in parts 1 and 2, I used the fuzzer described in the SSLL Testing Strategy. Particular attention was paid to writing fuzzer oerations which would force the list to grow and shrink so that the reallocation code paths would be exercised.

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 Testing Strategy

Paul Nickerson

Fuzz-testing

To test the Chained Dynamic Array-based List in parts 1 and 2, I used the fuzzer described in the SSLL Testing Strategy. Particular attention was paid to writing fuzzer operations which grew the list and then targeted items in the middle of the list in an effort to disrupt the CDAL mechanism that traverses across the chain and down the item array. In addition, copy operations were considered to be a potential source of bugs, so those code paths were targeted as well.

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. Compile
with ./compile.sh and run with ./fuzzer
=========================================================================

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

```
#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();
            }
```

```
47              return n;
48          }
49          Node* design_new_node(const T& element, Node* next = nullptr, bool dummy =
                false) {
50              Node* n = allocate_new_node();
51              n->is_dummy = dummy;
52              n->item = element;
53              n->next = next;
54              return n;
55          }
56          Node* design_new_node(Node* next = nullptr, bool dummy = false) {
57              Node* n = allocate_new_node();
58              n->is_dummy = dummy;
59              n->next = next;
60              return n;
61          }
62          void init() {
63              num_items = 0;
64              try {
65                  tail = design_new_node(nullptr, true);
66                  head = design_new_node(tail, true);
67              } catch (std::bad_alloc& ba) {
68                  std::cerr << "init(): failed to allocate memory for head/tail nodes"
                        << std::endl;
69                  throw std::bad_alloc();
70              }
71          }
72          //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)
73          void insert_node_after(Node* existing_node, Node* new_node) {
74              existing_node->next = new_node;
75              ++num_items;
76          }
77          //destroys the subsequent node and returns its item
78          T remove_item_after(Node* preceeding_node) {
79              Node* removed_node = preceeding_node->next;
80              T item = removed_node->item;
81              preceeding_node->next = removed_node->next;
82              delete removed_node;
83              --num_items;
84              return item;
85          }
86          void copy_constructor(const SSLL& src) {
87              const_iterator fin = src.end();
88              for (const_iterator iter = src.begin(); iter != fin; ++iter) {
89                  push_back(*iter);
90              }
91              if ( ! src.size() == size())
92                  throw std::runtime_error("copy_constructor: Copying failed - sizes
                        don't match up");
93          }
```

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

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

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

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

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

```
339         */
340         void push_back(const T& element) {
341             Node* new_tail;
342             try {
343                 new_tail = design_new_node(nullptr, true);
344             } catch (std::bad_alloc& ba) {
345                 std::cerr << "push_back(): failed to allocate memory for new tail"
                        << std::endl;
346                 throw std::bad_alloc();
347             }
348             insert_node_after(tail, new_tail);
349             //transform the current tail node from a dummy to a real node holding
                    element
350             tail->is_dummy = false;
351             tail->item = element;
352             tail->next = new_tail;
353             tail = tail->next;
354         }
355
356         /*
357             removes and returns the element at the list's head.
358         */
359         T pop_front() {
360             if (is_empty()) {
361                 throw std::out_of_range("pop_front: Can't pop: list is empty");
362             }
363             if (head->next == tail) {
364                 throw std::runtime_error("pop_front: head->next == tail, but list
                        says it's not empty (corrupt state)");
365             }
366             return remove_item_after(head);
367         }
368
369         //-------------------------------------------------
370         /*
371             removes and returns the element at the list's tail.
372         */
373         T pop_back() {
374             if (is_empty()) {
375                 throw std::out_of_range("pop_back: Can't pop: list is empty");
376             }
377             if (head->next == tail) {
378                 throw std::runtime_error("pop_back: head->next == tail, but list
                        says it's not empty (corrupt state)");
379             }
380             //XXX this is O(N), a disadvantage of this architecture
381             Node* node_before_last = node_before(size() - 1);
382             T item = remove_item_after(node_before_last);
383             return item;
384         }
385
386         //-------------------------------------------------
```

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

```
482                 out << "]";
483             }
484             return out;
485         }
486     }; //end class SSLL
487 } // end namespace cop3530
488 #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. Compile
with ./compile.sh and run with ./fuzzer
=========================================================================

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

psll/source/PSLL.h

PSLL.h

```
#ifndef _PSLL_H_
#define _PSLL_H_

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

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

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

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

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

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

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

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

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

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

```
385         */
386         void push_back(const T& element) {
387             Node* new_tail;
388             try {
389                 new_tail = design_new_node(nullptr, true);
390             } catch (std::bad_alloc& ba) {
391                 std::cerr << "push_back(): failed to allocate memory for new tail"
                        << std::endl;
392                 throw std::bad_alloc();
393             }
394             insert_node_after(tail, new_tail, num_main_list_items);
395             //transform the current tail node from a dummy to a real node holding
                    element
396             tail->is_dummy = false;
397             tail->item = element;
398             tail->next = new_tail;
399             tail = tail->next;
400         }
401
402         //-------------------------------------------------
403         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
404         //of that so the client still has a valid instance if our destructor is
                called
405         /*
406             removes and returns the element at the list's head.
407         */
408         T pop_front() {
409             if (is_empty()) {
410                 throw std::out_of_range("pop_front: Can't pop: list is empty");
411             }
412             if (head->next == tail) {
413                 throw std::runtime_error("pop_front: head->next == tail, but list
                        says it's not empty (corrupt state)");
414             }
415             return remove_item_after(head);
416         }
417
418         //-------------------------------------------------
419         /*
420             removes and returns the element at the list's tail.
421         */
422         T pop_back() {
423             if (is_empty()) {
424                 throw std::out_of_range("pop_back: Can't pop: list is empty");
425             }
426             if (head->next == tail) {
427                 throw std::runtime_error("pop_back: head->next == tail, but list
                        says it's not empty (corrupt state)");
428             }
429             //XXX this is O(N), a disadvantage of this architecture
430             Node* node_before_last = node_before(size() - 1);
```

```
431             T item = remove_item_after(node_before_last);
432             return item;
433         }
434 
435         //-------------------------------------------------
436         /*
437             removes and returns the the element at the specified position,
438             shifting the subsequent elements one position to the left.
439         */
440         T remove(size_t position) {
441             T item;
442             if (position >= size()) {
443                 throw std::out_of_range(std::string("remove: No element at position
                        ") + std::to_string(position));
444             }
445             if (head->next == tail) {
446                 throw std::runtime_error("remove: head->next == tail, but list says
                        it's not empty (corrupt state)");
447             }
448             //using a dummy head node guarantees that there be a node immediately
                    preceeding the specified position
449             Node *node_before_position = node_before(position);
450             item = remove_item_after(node_before_position);
451             return item;
452         }
453         //-------------------------------------------------
454         /*
455             returns (without removing from the list) the element at the specified
                    position.
456         */
457         T item_at(size_t position) const {
458             if (position >= size()) {
459                 throw std::out_of_range(std::string("item_at: No element at position
                        ") + std::to_string(position));
460             }
461             return operator[](position);
462         }
463 
464 
465         //-------------------------------------------------
466         /*
467             returns true IFF the list contains no elements.
468         */
469         bool is_empty() const {
470             return size() == 0;
471         }
472 
473         //-------------------------------------------------
474         /*
475             returns the number of elements in the list.
476         */
477         size_t size() const {
```

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

```
527                         out << ",";
528                     out << *iter;
529                 }
530                 out << "]";
531             }
532             return out;
533         }
534     }; //end class PSLL
535 } // end namespace cop3530
536 #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. Compile
with ./compile.sh and run with ./fuzzer
=========================================================================

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

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

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

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

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

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

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

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

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

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

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

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

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

cdal/source/CDAL.h

CDAL.h

```
#ifndef _CDAL_H_
#define _CDAL_H_

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

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

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

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

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

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

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

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

```
294         // types
295         //-----------------------------------------------
296         typedef CDAL_Iter iterator;
297         typedef CDAL_Const_Iter const_iterator;
298         typedef T value_type;
299
300         iterator begin() {
301             ItemLoc start_loc = loc_from_pos(0);
302             ItemLoc end_loc = loc_from_pos(size());
303             return iterator(start_loc, end_loc);
304         }
305
306         iterator end() {
307             ItemLoc end_loc = loc_from_pos(size());
308             return iterator(end_loc, end_loc);
309         }
310
311         const_iterator begin() const {
312             ItemLoc start_loc = loc_from_pos(0);
313             ItemLoc end_loc = loc_from_pos(size());
314             return const_iterator(start_loc, end_loc);
315         }
316
317         const_iterator end() const {
318             ItemLoc end_loc = loc_from_pos(size());
319             return const_iterator(end_loc, end_loc);
320         }
321
322         T& operator[](size_t i) {
323             if (i >= size()) {
324                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
325             }
326             return loc_from_pos(i).item_ref;
327         }
328
329         const T& operator[](size_t i) const {
330             if (i >= size()) {
331                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
332             }
333             return loc_from_pos(i).item_ref;
334         }
335
336         //-----------------------------------------------
337         // Constructors/destructor/assignment operator
338         //-----------------------------------------------
339
340         CDAL() {
341             init();
342             embiggen_if_necessary();
343         }
```

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

```
392             originally at that and those in subsequent positions one position to the
                    right.
393         */
394         void insert(const T& element, size_t position) {
395             if (position > size()) {
396                 throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
397             } else {
398                 embiggen_if_necessary();
399                 ItemLoc loc = loc_from_pos(position);
400                 //shift remaining items to the right
401                 T item_to_insert = element;
402                 Node* n = loc.node;
403                 for (size_t i = position; i <= num_items; ++i) {
404                     size_t array_index = i % array_size;
405                     if ( i != position && array_index == 0 ) {
406                         n = n->next;
407                     }
408                     std::swap(item_to_insert, n->item_array[array_index]);
409                 }
410                 ++num_items;
411             }
412         }
413
414         //-------------------------------------------------
415         //Note to self: use reference here because we receive the original object
                instance,
416         //then copy it into n->item so we have it if the original element goes out
                of scope
417         /*
418             prepends the specified element to the list.
419         */
420         void push_front(const T& element) {
421             insert(element, 0);
422         }
423
424         //-------------------------------------------------
425         /*
426             appends the specified element to the list.
427         */
428         void push_back(const T& element) {
429             insert(element, size());
430         }
431
432         //-------------------------------------------------
433         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
434         //of that so the client still has a valid instance if our destructor is
                called
435         /*
436             removes and returns the element at the list's head.
437         */
```

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

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

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