

SSL Informal Documentation

Paul Nickerson

List Methods

iterator begin()

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

iterator end()

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

const_iterator begin() const

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

const_iterator end() const

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

T& operator

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

const T& operator const

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

SLL(const SLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to `push_back()` each source item into the current list
- Afterwards, `this->size()` should equal `src.size()`

SLL& operator=(const SLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninitialized state, 2) initialize the class, and 3) use an iterator to `push_back()` each source item into the current list
- Returns a reference to `*this`, the copied-to instance
- Afterwards, `this->size()` should equal `src.size()`

T replace(const T& element, size_t position)

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

void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1

- May be called with a position one past the last stored item, in which case the new item becomes the last
 - In this case we pass the element to `push_back()`, which can do $O(1)$ insert
 - For position $< \text{size}()$, we do a $O(N)$ traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to `stderr` and `std::bad_alloc` is thrown

void push_front(const T& element)

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

void push_back(const T& element)

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

T pop_front()

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

T pop_back()

- Removes the node at position $(\text{size}() - 1)$, returning its stored item
- Points `preceding_node->next` to the tail
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a `runtime_error` if, after checking that the list is non-empty and prior to popping, `head->next == tail`. This would indicate internal list state corruption.

T remove(size_t position)

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

T item_at(size_t position) const

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

bool is_empty() const

- Returns true IIF size() == 0

size_t size() const

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

void clear()

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

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

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

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

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

Iterator Methods

explicit SLL_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

SLL_Iter(const SLL_Iter& src)

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

reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change

pointer operator->() const

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

self_reference operator=(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

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

bool operator!=(const self_type& rhs) const

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

Const Iterator Methods

explicit SLL_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

SLL_Const_Iter(const SLL_Const_Iter& src)

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

reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The `const` keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile

pointer operator->() const

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

self_reference operator==(const self_type& src)

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

self_reference operator++()

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

self_type operator++(int)

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

bool operator==(const self_type& rhs) const

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

bool operator!=(const self_type& rhs) const

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