

SSLLIter

```
// construct a SSLL Iterator assigning Node* here to NULL unless a different value is passed.
explicitSSLL_Iter(nodePtr start = NULL) : here(start) {}

// construct a SSLL iterator that sets Node* here to the same reference as src.here
SSLL_Iter(constSSLL_Iter&src) : here(src.here) {}

// return the data at the current location of the iterator if data does not exist, throw an
// out_of_range exception
reference operator*() const {}

// return a reference to the current location of the iterator if data does not exist, throw an
// out_of_range exception
pointer operator->() const {}

// if it's not self-reference, change Node* here to src.here.
self_reference operator=(constSSLL_Iter&src) {}

// increment the iterator by simply moving here to the next node in the list
self_reference operator++() {}

// increment the iterator by simply moving here to the next node in the list, but return the iterator
// with the state of the iterator before incrementing
self_type operator++(int) {}

// return whether or not the iterators point to the same thing.
bool operator==(constSSLL_Iter&rhs) const {}

// return whether or not the iterators point to the same thing.
bool operator!=(constSSLL_Iter&rhs) const {}
```

SSLL ConstIter

```
// construct a SSLL Iterator assigning Node* here to NULL unless a different value is passed. If //
valid, set it to thatvalue, else throw an invalid_argument exception
explicitSSLL_Const_Iter(nodePtr start) : here(start) {}

// construct a SSLL iterator that sets Node* here to the same reference as src.here
SSLL_Const_Iter(constSSLL_Const_Iter&src) : here(src.here) {}

// return the data at the current location of the iteratorif data does not exist, throw an
// out_of_range exception
reference operator*() const {}
```

```

// return a reference to the current location of the iterator if data does not exist, throw an
// out_of_range exception
pointer operator->() const { }

// if it's not self-reference, change Node* here to src.here.
self_reference operator=(constSLL_Const_Iter&src) { }

// increment the iterator by simply moving here to the next node in the list
self_reference operator++() { }

// increment the iterator by simply moving here to the next node in the list, but return the iterator
// with the state of the iterator before incrementing
self_type operator++(int) { }

// return whether or not the iterators point to the same thing.
bool operator==(constSLL_Const_Iter&rhs) const { }

// return whether or not the iterators point to the same thing.
bool operator!=(constSLL_Const_Iter&rhs) const { }

```

SLL

```

// constructor. Set head and tail to null. Set size to 0
SLL() { }

// copy constructor. Make hard copy of src
SLL(const SLL&src) { }

// destructor
~SLL() { }

// assignment operator overload. If not self-assignment, make hard copy of src's list
SLL& operator=(SLL&src) { }

// replace data in node at 'position' with 'element' if position is not valid, throw an
// invalid_argument exception
T replace(const T& element, int position) { }

// insert a node with data 'element' in 'position' push all nodes at current 'position' and after by 1
// throw invalid_argument exception if 'position' is invalid
void insert(const T& element, int position) { }

// insert a node with data 'element' to front of list
void push_front(const T& element) { }

```

```

// insert a node with data 'element' to end of list
void push_back(const T& element) {}

// remove the first node in the list and return the data it contained
T pop_front() {}

// remove the last node in the list and return the data it contained
T pop_back() {}

// remove the node at position 'position' and patch up the list to reflect this change. Throw an
// invalid_argument exception if 'position' is not a valid position in the list
T remove(int position) {}

// return the data at node 'position' without removing it throw invalid_argument if position is out
// of bounds
T item_at(int position) const {}

// create an iterator who's Node* here points to first element in list
iterator begin() {}

// create an iterator who's Node* here points to NULL (one past last element in list)
iterator end() {}

// create a const iterator who's Node* here points to first element in list
const_iterator begin() const {}

// create a const iterator who's Node* here points to NULL (one past last element in list)
const_iterator end() const {}

// return if the list is empty or not (if head points to anything or not)
bool is_empty() const {}

// clear the list by deleting every node in the list then setting head and tail to NULL and list size
// to 0
void clear() {}

// check if there exists a Node in the list who's data is the same as 'element' using the 'equals'
// method for comparison
bool contains(const T& element, bool equals(const T& a, const T& b)) const {}

// return size of the list
size_t size() const {}

// return an ostream& containing a print of the list's elements
std::ostream& print(std::ostream& out) const {}

```

```
// return a reference to the T data in node at 'position' throw invalid_argument exception if  
// position is invalid  
T&operator[](int position) { }
```

```
// return a const reference to the T data in node at 'position' throw invalid_argument exception if  
// position is invalid  
T const& operator[](int position) const { }
```