SDAL Iter

```
// construct a SDAL Iterator assigning T* here to NULL unless a parameter is passed.
explicitSDAL_Iter(pointer start) : here(start) {}

// construct a SDAL iterator that sets T* here to the same reference as src.here
SDAL_Iter(constSDAL_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 T* here to src.here.
self_reference operator=(constSDAL_Iter&src) {}

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

// increment the iterator by simply moving here to the next element in the array, 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==(constSDAL_Iter&rhs) const {}

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

SDAL Const_Iter

```
// construct a SDAL Iterator assigning T* here to NULL unless a parameter is passed.
explicitSDAL_Const_Iter(pointer start) : here(start) {}

// construct a SDAL iterator that sets T* here to the same reference as src.here
SDAL_Const_Iter(constSDAL_Const_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 T* here to src.here.
self_reference operator=(constSDAL_Const_Iter&src) {}

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

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

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

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

SDAL

```
// default contructor. Set head and tail to -1, array size to 50, and list size to 0
SDAL() {}

// constructor set appropriate size
SDAL(inta_size) {}

// copy constructor, make hard copy of src.
SDAL(const SDAL&src) {}

// destructor
~SDAL() {}

// assignment operator overload. If not self-assignment, clear contents and made hard copy of
// src's array
SDAL& operator=(const SDAL&src) {}

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

```
// first update list in case it is full insert data 'element' in array slot 'position' push all elements at
// current 'position' and after by 1 and increase list size by 1 throw invalid_argument exception if
// 'position' is invalid
void insert(const T& element, int position) {}

// remove the element at position 'position' and patch up the list to reflect this change. Update list
// after removal according to stated conditions. Throw an invalid_argument exception if 'position'
// is not a valid position in the list
T remove(int position) {}

// insert 'element' to front of array if element is not type T, throw invalid_argument exception
voidpush_front(const T& element) {}

// insert 'element' to end of array if element is not type T, throw invalid_argument exception
voidpush_back(const T& element) {}

// remove the first element in the array and return it
T pop_front() {}

// remove the last element in the array and return it
T pop_back() {}

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

// create an iterator who's T* here points to first element in the array
iterator begin() {}

// create an iterator who's T* here points to one past last element in the array
iterator end() {}

// create a const iterator who's T* here points to first element in the array
const_iterator begin() const {}

// create a const iterator who's T* here points to one past last element in the array
const_iterator end() const {}

// clear the list by setting every element to NULL and setting head & tail = -1 and list_size to 0
// and update list according to conditions
void clear() {}

// return if list is empty or not (if head = -1)
boolis_empty() const {}

// return size of list
```

```
size_t size() const {}

// check if there exists an element in the array equal to 'element' using the 'equals' method for
// comparison
bool contains(const T& element, bool equals(const T& a, constT&b)) const {}
// update list method: if insert is called and array is full, allocate new array of size 150% original.
// if array size>99 and list size < half of array size
// allocate new array of size 50% original. Then copy over elements and deallocate original
voidupdate_list() {}

// 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 array at 'position' throw invalid_argument exception if
// position is invalid
T&operator[](int position) {}

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