CDAL Iter

```
// construct a CDAL Iterator assigning T* here to NULL and index to 0 unless a parameter is
// passed.
explicitCDAL_Iter(nodePtr start = NULL) : here(start) {}

// construct a SDAL iterator that sets T* here to the same reference as src.here and index to
// src.index
CDAL_Iter(constCDAL_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 and index to src.index.
self_reference operator=(constCDAL_Iter&src) {}

// increment the iterator by simply moving here to the next element in the array. If reached end of
// array and there is a next node, change here to next node and index to 0
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. If reached end of array andthere is a
// next node, change here to next node and index to 0
self_type operator++(int) {}

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

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

CDAL Const_Iter

```
// construct a CDAL Iterator assigning T* here to NULL and index to 0 unless a parameter is
// passed.
explicitCDAL_Const_Iter(pointer start = NULL) : here(start) {

// construct a SDAL iterator that sets T* here to the same reference as src.here and index to
// src.index
```

```
CDAL_Const_Iter(constCDAL_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 iteratorif 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 and index to src.index.
self_reference operator=(constCDAL_Const_Iter&src) {}

// increment the iterator by simply moving here to the next element in the array. If reached end of
// array and there is a next node, change here to next node and index to 0
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. If reached end of array and there is a
// next node, change here to next node and index to 0
self_type operator++(int) {}

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

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


CDAL




// constructor
CDAL() {}

// make hard copy of src's data
CDAL(const CDAL&src) {}

// destroy the list by deleting every node in the list then set instance variables to appropriate
// values
~CDAL() {}
```

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

// find the appropriate node in list where position is and insert 'element' there. Then patch list to
// reflect change and update list size and tail. 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_argumentexception if 'position'
// is not a valid position in the list
T remove(int position) {}

// 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() {}

// 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) {}

// 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 Node* here points to first node in the 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 node in the 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 list is empty or not (if head = -1)
boolis_empty() const{}

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

// clear the list by setting all elements in all arrays to NULL then update list according to
// conditions
void clear() {}

// update list method: if more than half of the arrays are unuseddeallocate half the unused items
voidupdate_list() {}

// check if there exists an element in the arrays equal to 'element' using the 'equals' method for
// comparison
bool contains(const T& element, bool equals(const T& a, const T& b)) 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 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 {}
```