Chained Dynamic Array-Based List
CDAL

CDAL()
Constructor for the CDAL. It simply creates an instance of the class and sets the head and and tail equal to NULL. It also initializes the count (amount of items in the list to zero) and the amount of nodes in the list to zero. The class also uses an index to check where to insert.

CDAL(const CDAL& src)
Copy constructor.

~CDAL()
Default destructor for the class

void CDAL<T>::indexConvert()
Private function used to convert the index back to zero when it reaches 50.

void CDAL<T>::shift(node *current)
Private function used to shift all of the elements past a certain point in the CDAL. It takes a pointer to a node and shifts everything after that forward.

void CDAL<T>::insertHelper(node* current, int position, int nodeNumber)
This private method takes a pointer to a node, a position, and a node number. It is used as a helper to the class' insert method.

void CDAL<T>::allocNode(T e)
This method is used to allocate a new node in the list.

T replace(const T &e, int position)
This method takes an element of type T and a position and replaces the value at the given position with a new one. If the user tries to pass a position that is not valid, then an out of range exception is thrown. Positions have to range within position 0 and the max size of the list.

void CDAL<T>::insert(const T &e, int position)
This method takes an element of type T and a position and inserts a value at the given position. The method only accepts positions that are within position 0 and the size of the list. This means that an insert can occur one after the last element, just as if the user was push_back(). If the position does not fall within this range, then an out of range exception is thrown. Once an element is inserted at the beginning or in the middle of the list then all of the elements have to be shifted.

void CDAL<T>::push_front(const T &e)
This method inserts an element at the beginning of the list and shifts everything else one index up. If the list is "full" at the time of pushing then a new node is added to the back of the list

void CDAL<T>::push_back(const T &e)
This method inserts an element at the end of the list. If the list is full at the time of pushing, then a new node is added and the new node would then contain the value that was passed in through this method.

T CDAL<T>::pop_front()
This method removes the element at the beginning of the list and returns it. It then shifts all of the elements one down. If the user tries to pop from an empty list then the program will throw a logic error, telling the user that the list is empty and that they are not able to pop from it. If the popping of this element results in the zeroing out of an array, then the array will be deallocated.

T CDAL<T>::pop_back()
This method removes the last element from the list and returns the value that was stored in it. Just as pop_front(), this method also throws a logic error if the user tries to pop from an empty list. If the popping of this element results in the zeroing out of an array, then the array will be deallocated.

T CDAL<T>::remove(int position)
This method takes an integer position and removes the item at that position. It also returns the value that was stored in that node. If the list is empty when the remove method is called, then a logic error will be thrown at the user. If the given integer position is not within 0 and the list's size, then an out of range exception will be thrown to the user. If the removal of this element results in the zeroing out of an array, then the array will be deallocated.

T CDAL<T>::itemAt(int position)
This method takes an integer position and returns the data of the element at that position. Just like the other methods that take an integer position, if the given position is not within the boundaries of the list, then an out of range exception is thrown telling the user that the position they entered was invalid.

bool CDAL<T>::isEmpty() const
This method checks if the list is empty. It returns true if the list is empty and false otherwise.

int CDAL<T>::size() const
This method returns the current size of the list.

void CDAL<T>::clear()
This method empties out the list.

bool CDAL<T>::contains(T e, bool equals_function)
This method checks to see if a given element value is in the list. It returns true if it is and false if it is not.

const ostream& CDAL<T>::print(ostream& out)
This method prints out the list. It stores the result in the ostream.

CDAL<T>::CDAL_ITER CDAL<T>::begin()
This method returns an iterator to the beginning of the list

CDAL<T>::CDAL_ITER CDAL<T>::end()
This method returns an iterator to the end of the list.

CDAL<T>::CDAL_Const_ITER CDAL<T>::begin() const
This method returns a constant iterator to the beginning of a constant list.

CDAL<T>::CDAL_Const_ITER CDAL<T>::end() const
This method returns a constant iterator to the end of a constant list.

T& operator[](int i)
This operator overloading method lets the list be indexed like an array even though it is a linked list.

T const& operator[](int i)const
This operator overloading method lets the constant list be indexed like an array even though it is a linked list.

CDAL_ITER
explicit CDAL_ITER(node *n)
Explicit constructor

CDAL_ITER(const CDAL_ITER& src)
Copy constructor

pointer operator->()
Returns the data stored in what the iterator is pointing at.

reference operator *()
Returns the data stored in what the iterator is pointing at.

self_reference operator++()
Pre-increment operator overloading.

self_type operator++(int)
post-increment operator overloading

bool operator ==(CDAL_ITER &rhs)
Equality operator overloading. Compares to see if two iterators are the same.

bool operator!=(CDAL_ITER &rhs)
Inequality operator overload. Compares to see if two iterators are not the same.

self_reference operator=(CDAL_ITER &rhs)
Assignment operator.

ostream &operator<<(ostream &obj)
Allows for iterators to print the data that they are pointing at.

CDAL_Const_ITER
The following are the methods in the CDAL_Const_ITER class. Their descriptions are the same as the preceding class except that these overloads are for a constant iterator.

explicit CDAL__ConstITER(node *n)
CDAL_ITER(const CDAL_Const_ITER& src)
pointer operator->()
reference operator *()
self_reference operator++()
self_type operator++(int)
ool operator ==(CDAL_Const_ITER &rhs)
bool operator!=(CDAL_Const-ITER &rhs)
self_reference operator=(CDAL_Const_ITER &rhs)
ostream &operator<<(ostream &obj)