ECE 250 Algorithms and Data Structure
Project One: Double Sentinel List

Soheil Soltani

September 13, 2018

UNIVERSITY OF WATERLOO

Constructor

```
//Constructor
template <typename Type>
Double_sentinel_list<Type>::Double_sentinel_list():
list_head(nullptr),
list_tail(nullptr),
list_size( 0 )
{
// Enter your implementation here
//create the head sentinel node

//Get the list_head pointer to point to the head sentinel node

// Create the head sentinel node and using the tail sentinel to point back to head
sentinel and also to nullptr

//Get head sentinel node to point to tail sentinel

//Get the list_tail pointer to point back to the tail sentinel

}
```

Copy Constructor

```
//Copy Constructor
template <typename Type>
Double_sentinel_list<Type>::Double_sentinel_list( Double_sentinel_list<Type> const &list ):
// Updated the initialization list here
list_head( nullptr ),
list_tail( nullptr ),
list_size( 0 )

{
// Enter your implementation here

//create the empty list for the copy constructor list similar to constructor

//If the original list is empty, no need to copy anything

//A loop having both original list and new list traverse and import values

//Set each next node to a new node and its necessary relations to previous and next node

//Increment the list size
}
```

Move Constructor

```
//Move Constructor
template <typename Type>
Double_sentinel_list<Type>::Double_sentinel_list( Double_sentinel_list<Type> &&list ):
// Updated the initialization list here
list_head( nullptr ),
list_tail( nullptr ),
list_size( 0 )
{
// Enter your implementation here
//same to the simple constructor


//Call the list swap function which will move the contents of the list passed in as the argument
  into the new list created in this constructor


//Get the original sentinels to point to each other and list_size to zero


}
```

Destructor

- //Destructor
- template <typename Type>
- Double_sentinel_list<Type>::~Double_sentinel_list() {
- // Enter your implementation here
- //Delete (Pop) all the nodes until only the sentinels are left
- //Delete the pointers as well to completely get rid of everything
-
- }

List Size

- //Returns how many items is in the list
- template <typename Type>
- int Double_sentinel_list<Type>::size() const {
- // Enter your implementation here
- //There is a list_size counter implemented throughout

- //The most up to date value is returned
- }

empty()

```
//Returns true if the list is empty otherwise false
template <typename Type>
bool Double_sentinel_list<Type>::empty() const {
//Enter your implementation here
//Just check if the list is empty

}
```

front()

- //This will return the contents of the node that the head sentinel points to
- template <typename Type>
- Type Double_sentinel_list<Type>::front() const {
- //Enter your implementation here
- //If the list is empty throw underflow

- //Return the value of the first node that the head sentinel points to
- }

back()

- //Returns the contents stored in the node that the prev of tail sentinel
- template <typename Type>
- Type Double_sentinel_list<Type>::front() const {
- //Enter your implementation here
- //If the list is empty throw underflow

- //Return the value of the last node in the linked list before tail sentinel
- }

begin()

- //Returns the address of what the head sentinel node points to
- template <typename Type>
- typename Double_sentinel_list<Type>::Double_node *Double_sentinel_list<Type>::begin() const {
- //Enter your implementation here
- }

end()

```
//Returns the address of the tail sentinel itself
template <typename Type>
typename Double_sentinel_list<Type>::Double_node
*Double_sentinel_list<Type>::end() const {
//Enter your implementation here

}
```

rbegin()

```
//Returns the address of what tail sentinel is pointing to
template <typename Type>
typename Double_sentinel_list<Type>::Double_node *Double_sentinel_list<Type>::rbegin() const {
//Enter your implementation here

}
```

rend()

```
• //Returns the address of the head sentinel node itself
• template <typename Type>
• typename Double_sentinel_list<Type>::Double_node
  *Double_sentinel_list<Type>::rend() const {
• // Enter your implementation here

• }
```

find()

- //Finds the first occurrence of the passed obj and returns the address
- template <typename Type>
- typename Double_sentinel_list<Type>::Double_node *Double_sentinel_list<Type>::find(Type const &obj) const {
- //Enter your implementation here
- //Iterate through the list
- //If it sees a node content matches the obj, return the address of the node
- //If found no match, return the list tail
- }

count()

- //Find how many times the passed obj is found
- template <typename Type>
- int Double_sentinel_list<Type>::count(Type const &obj) const {
- //Enter your implementation here
- //Iterate through the list
- //If it sees a node content matches the obj, increment node_count
- }

Push_front()

```
•  //Put in a new node at the front of the list
•  template <typename Type>
•  void Double_sentinel_list<Type>::push_front( Type const
   &obj ) {
•  //Initialize a new node

•  //Have the previous of the next of head sentinel node to
   be the new node

•  //Have the next of head be this new node

•  //Increment size

•  }
```

push_back()

```
//Put in a new node at the end of the list(before sentinel)
template <typename Type>
void Double_sentinel_list<Type>::push_back( Type const &obj ) {
//Initialize a new node

//Have the next of the previous of tail sentinel node to be the
new node

//Have the previous of tail be this new node


//Increment size


}
```

pop_front()

```
//Removes the first node in the list
template <typename Type>
void Double_sentinel_list<Type>::pop_front() {
//Throw underflow exception when list is empty

//Initialize a dummy node and equal to the begin node

//Re-reference the previous node of the 2nd node to list head

//Re-reference the next node of list head to the 2nd node

//Clean up


//Decrement size

}
```

pop_back()

```
//Removes the last node in the list
template <typename Type>
void Double_sentinel_list<Type>::pop_back() {
//Throw underflow exception when list is empty

//Initialize a dummy node and equal to the begin node

//Re-reference the next node of the 2nd last node to list tail

//Re-reference the previous node of list tail to the 2nd last node

//Clean up


//Decrement size

}
```

erase

- //Erases all instances of the obj that can be found in the list
- int Double_sentinel_list<Type>::erase(Type const &obj) {
- //Initialize a counter
- int node_count = 0;
- //Iterate through the list

- //If it finds a match, increment the counter, delete that node

- //Return the counter

- }

Double_node Constructor

```
template <typename Type>
Double_sentinel_list<Type>::Double_node::Double_node(
Type const &nv,
typename Double_sentinel_list<Type>::Double_node *pn,
typename Double_sentinel_list<Type>::Double_node *nn ):

node_value( Type() ), // This assigns 'node_value' the default value of Type
previous_node(nullptr ),
next_node(nullptr )
{
// Enter your implementation here
// Initialize the attributes of the node with input parameters

}
```