

## A Generic Linked List

Generated by Doxygen 1.9.7



<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Class Documentation</b>	<b>5</b>
3.1 LinkedList< T > Class Template Reference	5
3.1.1 Detailed Description	5
3.1.2 Constructor & Destructor Documentation	6
3.1.2.1 LinkedList()	6
3.1.2.2 ~LinkedList()	6
3.1.3 Member Function Documentation	6
3.1.3.1 add()	6
3.1.3.2 addFromFile()	7
3.1.3.3 binarySearch()	7
3.1.3.4 bubbleSort()	8
3.1.3.5 clear()	8
3.1.3.6 insert()	9
3.1.3.7 mergeLists()	9
3.1.3.8 mergeSort()	10
3.1.3.9 print()	11
3.1.3.10 remove()	11
3.1.3.11 search()	12
3.1.3.12 toString()	12
3.2 Node< T > Class Template Reference	13
3.2.1 Detailed Description	13
3.2.2 Constructor & Destructor Documentation	13
3.2.2.1 Node() [1/2]	13
3.2.2.2 Node() [2/2]	13
3.2.2.3 ~Node()	14
3.2.3 Member Function Documentation	14
3.2.3.1 getData()	14
3.2.3.2 getNextNode()	14
3.2.3.3 getPrevNode()	14
3.2.3.4 setData()	15
3.2.3.5 setNextNode()	15
3.2.3.6 setNextNodeNull()	15
3.2.3.7 setPrevNode()	15
3.2.3.8 setPrevNodeNull()	15
3.3 Vault Class Reference	16
3.3.1 Detailed Description	16
3.3.2 Constructor & Destructor Documentation	16

---

3.3.2.1 Vault() [1/2]	16
3.3.2.2 Vault() [2/2]	16
3.3.2.3 ~Vault()	16
3.3.3 Member Function Documentation	17
3.3.3.1 operator!=(())	17
3.3.3.2 operator<()	17
3.3.3.3 operator<=()	17
3.3.3.4 operator==(())	17
3.3.3.5 operator>()	17
3.3.3.6 operator>=()	18
3.3.4 Friends And Related Symbol Documentation	18
3.3.4.1 operator<<	18
<b>4 File Documentation</b>	<b>19</b>
4.1 FunctionTests.hpp	19
4.2 LinkedList.cpp	25
4.3 LinkedList.hpp	29
4.4 main.cpp	30
4.5 Node.cpp	30
4.6 Node.hpp	31
4.7 Vault.cpp	31
4.8 Vault.hpp	32
<b>Index</b>	<b>33</b>

# Chapter 1

## Class Index

### 1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">LinkedList&lt; T &gt;</a>	<a href="#">5</a>
<a href="#">Node&lt; T &gt;</a>	<a href="#">13</a>
<a href="#">Vault</a>	<a href="#">16</a>



## Chapter 2

# File Index

### 2.1 File List

Here is a list of all documented files with brief descriptions:

<a href="#">FunctionTests.hpp</a>	19
<a href="#">LinkedList.cpp</a>	25
<a href="#">LinkedList.hpp</a>	29
<a href="#">main.cpp</a>	30
<a href="#">Node.cpp</a>	30
<a href="#">Node.hpp</a>	31
<a href="#">Vault.cpp</a>	31
<a href="#">Vault.hpp</a>	32





# Chapter 3

## Class Documentation

### 3.1 `LinkedList< T >` Class Template Reference

#### Public Member Functions

- `LinkedList ()`
- `~LinkedList ()`  
*Deconstructor.*
- `void clear ()`  
*This function removes all allocated memory used by Linked List.*
- `void insert (T data)`
- `void add (T data)`
- `void remove (T data)`
- `Node< T > * search (T data)`
- `std::string toString ()`
- `void mergeSort ()`
- `void bubbleSort ()`
- `void addFromFile (std::string fileName)`
- `void mergeLists (const LinkedList< T > *listTwo)`
- `void print ()`  
*This function prints each Node's data in Linked List to console.*
- `Node< T > * binarySearch (T target)`

#### 3.1.1 Detailed Description

```
template<class T>
class LinkedList< T >
```

Definition at line 16 of file `LinkedList.hpp`.

### 3.1.2 Constructor & Destructor Documentation

#### 3.1.2.1 LinkedList()

```
template<class T >
LinkedList< T >::LinkedList
```

Constructor

Definition at line 13 of file [LinkedList.cpp](#).

```
00013      {
00014      this->head = nullptr;
00015      this->tail = nullptr;
00016  }
```

#### 3.1.2.2 ~LinkedList()

```
template<class T >
LinkedList< T >::~~LinkedList
```

Deconstructor.

Definition at line 19 of file [LinkedList.cpp](#).

```
00019      {
00020      clear();
00021  }
```

### 3.1.3 Member Function Documentation

#### 3.1.3.1 add()

```
template<class T >
void LinkedList< T >::add (
    T data )
```

This function adds a [Node](#) containing the data passed

Parameters

<i>data</i>	- the data a <a href="#">Node</a> contains
-------------	--

Definition at line 66 of file [LinkedList.cpp](#).

```
00066      {
00067      Node<T>* newNode = new Node<T>(data);
00068
00069      if (this->head == nullptr) {
00070          this->head = newNode;
00071          this->tail = newNode;
00072      }
00073      else {
00074          this->tail->setNextNode(newNode);
00075          newNode->setPrevNode(tail);
00076          this->tail = newNode;
00077      }
00078  }
```

### 3.1.3.2 addFromFile()

```
template<class T >
void LinkedList< T >::addFromFile (
    std::string fileName )
```

This function adds data from a txt file into Linked List

#### Parameters

<i>fileName</i>	- name of the txt file. File name must include .txt extension
-----------------	---

Definition at line 268 of file [LinkedList.cpp](#).

```
00268                                     {
00269     std::ifstream file;
00270     T data;
00271
00272     file.open(fileName);
00273
00274     if (file.peek() == std::ifstream::traits_type::eof()) {
00275         std::cerr << "Error: File is empty" << std::endl;
00276         exit(1);
00277     }
00278
00279     if (!file.is_open()) {
00280         std::cerr << "Error opening file" << std::endl;
00281         exit(1);
00282     }
00283
00284     while (file >> data) {
00285         if (file.eof()) {
00286             break;
00287         }
00288
00289         this->add(data);
00290     }
00291
00292     if (!file.eof()) {
00293         std::cerr << "Error reaching end of file" << std::endl;
00294         exit(1);
00295     }
00296
00297     file.close();
00298 }
```

### 3.1.3.3 binarySearch()

```
template<class T >
Node< T > * LinkedList< T >::binarySearch (
    T target )
```

Searches for a value using binary search. Requires the list to be sorted to work.

#### Parameters

<i>target</i>	The value to look for.
---------------	------------------------

Definition at line 345 of file [LinkedList.cpp](#).

```
00345                                     {
00346     Node<T>* searchHead = this->head;
00347     Node<T>* searchTail = this->tail;
00348     Node<T>* searchMid = findMid(searchHead, searchTail);
00349     if (searchHead) {
00350         while (searchHead->getData() <= searchTail->getData()) {
00351             if (target == searchHead->getData()) {
00352                 return searchHead;
```

```

00353         }
00354         else if (target == searchMid->getData()) {
00355             return searchMid;
00356         }
00357         else if (target == searchTail->getData()) {
00358             return searchTail;
00359         }
00360
00361         if (target < searchMid->getData()) {
00362             searchHead = searchHead->getNextNode();
00363             searchTail = searchMid->getPrevNode();
00364             searchMid = findMid(searchHead, searchTail);
00365         }
00366         else if (target > searchMid->getData()) {
00367             searchHead = searchMid->getNextNode();
00368             searchTail = searchTail->getPrevNode();
00369             searchMid = findMid(searchHead, searchTail);
00370         }
00371     }
00372 }
00373
00374 return nullptr;
00375 }

```

### 3.1.3.4 bubbleSort()

```

template<class T >
void LinkedList< T >::bubbleSort

```

Sorts the [LinkedList](#) using the bubble sort algorithm.

Definition at line 239 of file [LinkedList.cpp](#).

```

00239     {
00240         // do not sort if empty or one
00241         if (this->head == nullptr || this->head->getNextNode() == nullptr)
00242             return;
00243
00244         bool swap;
00245         Node<T>* current = this->head;
00246         Node<T>* sorttail = nullptr;
00247
00248         while (current != sorttail){
00249             swap = false;
00250             Node <T>* current2 = this->head;
00251
00252             while (current2->getNextNode () != sorttail){
00253                 if (current2->getData() > current2->getNextNode()->getData()){
00254                     T temp = current2 -> getData();
00255                     current2->setData(current2->getNextNode()->getData());
00256                     current2->getNextNode()->setData(temp);
00257                     swap = true;
00258                 }
00259                 current2 = current2->getNextNode();
00260             }
00261             sorttail = current2; // update tail to last swap
00262             if (!swap)
00263                 break; // if no swap the list is already sorted
00264         }
00265     }

```

### 3.1.3.5 clear()

```

template<class T >
void LinkedList< T >::clear

```

This function removes all allocated memory used by [LinkedList](#).

Definition at line 24 of file [LinkedList.cpp](#).

```

00024     {
00025         Node<T>* nodeToDelete = head;
00026         while (head != nullptr) {
00027             head = head->getNextNode();
00028             delete nodeToDelete;
00029             nodeToDelete = head;
00030         }
00031     }

```

## 3.1.3.6 insert()

```
template<class T >
void LinkedList< T >::insert (
    T data )
```

This function inserts a [Node](#) that contains the specific data in Linked List in order

## Parameters

<i>data</i>	- the data a <a href="#">Node</a> contains
-------------	--

Definition at line 34 of file [LinkedList.cpp](#).

```
00034 {
00035     Node<T>* newNode = new Node<T>(data);
00036     if (this->head == nullptr) {
00037         this->head = newNode;
00038         this->tail = newNode;
00039         return;
00040     }
00041     if (data <= this->head->getData()) {
00042         newNode->setNextNode(head);
00043         this->head->setPrevNode(newNode);
00044         this->head = newNode;
00045         return;
00046     }
00047     if (data >= tail->getData()) {
00048         newNode->setPrevNode(tail);
00049         this->tail->setNextNode(newNode);
00050         this->tail = newNode;
00051         return;
00052     }
00053     Node<T>* temp = this->head;
00054     while (temp->getNextNode() != nullptr && temp->getNextNode()->getData() < data) {
00055         temp = temp->getNextNode();
00056     }
00057 }
00058
00059 newNode->setNextNode(temp->getNextNode());
00060 temp->setNextNode(newNode);
00061 newNode->setPrevNode(temp);
00062 newNode->getNextNode()->setPrevNode(newNode);
00063 }
```

## 3.1.3.7 mergeLists()

```
template<class T >
void LinkedList< T >::mergeLists (
    const LinkedList< T > * listTwo )
```

Modified the [LinkedList](#) from which it was called. Calling [LinkedList](#) will be modified and sorted.

## Parameters

<i>listTwo</i>	Does not need to be sorted and remains unchanged.
----------------	---

Definition at line 307 of file [LinkedList.cpp](#).

```
00307 {
00308     if (!this->head && !listTwo->head) {
00309         return;
00310     }
00311     else {
00312         mergeSort();
00313         Node<T>* temp = listTwo->head;
00314         while (temp != nullptr) {
00315             this->insert(temp->getData());
```

```

00316         temp = temp->getNextNode();
00317     }
00318 }
00319 }

```

### 3.1.3.8 mergeSort()

```

template<class T >
void LinkedList< T >::mergeSort

```

Sorts the [LinkedList](#) using the merge sort algorithm.

Definition at line 142 of file [LinkedList.cpp](#).

```

00142     {
00143         // base case: 1 or 0 Nodes
00144         if (this->head == nullptr || this->head->getNextNode() == nullptr) {
00145             return;
00146         }
00147
00148         // split the LinkedList in half
00149         Node<T>* subHead1 = this->head;
00150         Node<T>* subHead2 = findMid(this->head, this->tail);
00151         Node<T>* subTail1 = subHead2->getPrevNode();
00152         Node<T>* subTail2 = this->tail;
00153
00154         // detach the two halves
00155         subTail1->setNextNodeNull();
00156         subHead2->setPrevNodeNull();
00157
00158         // recurse first half
00159         this->head = subHead1;
00160         this->tail = subTail1;
00161         mergeSort();
00162         subHead1 = this->head;
00163         subTail1 = this->tail;
00164
00165         // recurse second half
00166         this->head = subHead2;
00167         this->tail = subTail2;
00168         mergeSort();
00169         subHead2 = this->head;
00170         subTail2 = this->tail;
00171
00172         // merge both halves
00173         this->head = nullptr;
00174         Node<T>* nodeptr = nullptr;
00175
00176         // compare head of both halves
00177         while (subHead1 != nullptr && subHead2 != nullptr) {
00178             if (subHead1->getData() < subHead2->getData()) {
00179                 if (this->head == nullptr) {
00180                     this->head = subHead1;
00181                     nodeptr = subHead1;
00182                 }
00183                 else {
00184                     nodeptr->setNextNode(subHead1);
00185                     subHead1->setPrevNode(nodeptr);
00186                     nodeptr = subHead1;
00187                 }
00188                 subHead1 = subHead1->getNextNode();
00189             }
00190             else {
00191                 if (this->head == nullptr) {
00192                     this->head = subHead2;
00193                     nodeptr = subHead2;
00194                 }
00195                 else {
00196                     nodeptr->setNextNode(subHead2);
00197                     subHead1->setPrevNode(nodeptr);
00198                     nodeptr = subHead2;
00199                 }
00200                 subHead2 = subHead2->getNextNode();
00201             }
00202         }
00203
00204         // add the rest of first half to the main LinkedList
00205         while (subHead1 != nullptr) {
00206             if (this->head == nullptr) {
00207                 this->head = subHead1;
00208                 nodeptr = subHead1;

```

```

00209     }
00210     else {
00211         nodeptr->setNextNode(subHead1);
00212         subHead1->setPrevNode(nodeptr);
00213         nodeptr = subHead1;
00214     }
00215     subHead1 = subHead1->getNextNode();
00216 }
00217
00218 // add the rest of second half to the main LinkedList
00219 while (subHead2 != nullptr) {
00220     if (this->head == nullptr) {
00221         this->head = subHead2;
00222         nodeptr = subHead2;
00223     }
00224     else {
00225         nodeptr->setNextNode(subHead2);
00226         subHead2->setPrevNode(nodeptr);
00227         nodeptr = subHead2;
00228     }
00229     subHead2 = subHead2->getNextNode();
00230 }
00231
00232 this->tail = nodeptr;
00233 }

```

### 3.1.3.9 print()

```

template<class T >
void LinkedList< T >::print

```

This function prints each [Node](#)'s data in Linked List to console.

Definition at line 322 of file [LinkedList.cpp](#).

```

00322     {
00323         Node<T>* temp = this->head;
00324
00325         if (this->head == nullptr) {
00326             std::cout << "The linked list is empty." << std::endl;
00327             return;
00328         }
00329
00330         while (temp != nullptr) {
00331             std::cout << temp->getData() << " ";
00332             temp = temp->getNextNode();
00333         }
00334
00335         std::cout << std::endl;
00336     }

```

### 3.1.3.10 remove()

```

template<class T >
void LinkedList< T >::remove (
    T data )

```

This function removes a [Node](#) from the Linked List containing the data passed

#### Parameters

<i>data</i>	- the data a <a href="#">Node</a> contains
-------------	--

Definition at line 81 of file [LinkedList.cpp](#).

```

00081     {
00082         Node<T>* nodeToDelete = search(data);
00083         if (nodeToDelete != nullptr) {
00084             if (nodeToDelete == this->head) {
00085                 this->head = head->getNextNode();

```

```

00086         this->head->setPrevNodeNull();
00087     }
00088     else if (nodeToDelete == this->tail) {
00089         this->tail = tail->getPrevNode();
00090         this->tail->setNextNodeNull();
00091     }
00092     else {
00093         Node<T>* prevNode = nodeToDelete->getPrevNode();
00094         Node<T>* nextNode = nodeToDelete->getNextNode();
00095         prevNode->setNextNode(nextNode);
00096         nextNode->setPrevNode(prevNode);
00097     }
00098     delete nodeToDelete;
00099 }
00100 }

```

### 3.1.3.11 search()

```

template<class T >
Node< T > * LinkedList< T >::search (
    T data )

```

This function searches for a [Node](#) containing the data passed and returns a reference to [Node](#)

#### Parameters

<i>data</i>	- the data a <a href="#">Node</a> contains
-------------	--

#### Returns

[Node](#) reference to [Node](#) that contains data passed, if not found, returns nullptr

Definition at line 103 of file [LinkedList.cpp](#).

```

00103     {
00104         Node<T>* temp = this->head;
00105     while (temp) {
00106         if (temp->getData() == data) {
00107             return temp;
00108         }
00109         temp = temp->getNextNode();
00110     }
00111     return nullptr;
00112 }
00113
00114
00115 }

```

### 3.1.3.12 toString()

```

template<class T >
std::string LinkedList< T >::toString

```

Returns the string representation of the [LinkedList](#) in the form of an array.

Definition at line 122 of file [LinkedList.cpp](#).

```

00122     {
00123         Node<T>* temp = this->head;
00124         std::string output = "";
00125         std::string quote = (typeid(T).name() == typeid(std::string("")).name()) ? "\"" : "";
00126     while (temp != nullptr) {
00127         T val = temp->getData();
00128         output += quote + to_string(val) + quote;
00129         temp = temp->getNextNode();
00130         if (temp != nullptr) {
00131

```



```

00132         output += ", ";
00133     }
00134 }
00135     return "{" + output + "}";
00136 }

```

The documentation for this class was generated from the following files:

- [LinkedList.hpp](#)
- [LinkedList.cpp](#)

## 3.2 Node< T > Class Template Reference

### Public Member Functions

- [Node](#) ()
- [Node](#) (T data)
- [~Node](#) ()
- T [getData](#) ()
- [Node](#) \* [getNextNode](#) ()
- [Node](#) \* [getPrevNode](#) ()
- void [setData](#) (T data)
- void [setNextNode](#) ([Node](#) \*next)
- void [setPrevNode](#) ([Node](#) \*prev)
- void [setNextNodeNull](#) ()
- void [setPrevNodeNull](#) ()

### 3.2.1 Detailed Description

```

template<class T>
class Node< T >

```

Definition at line 12 of file [Node.hpp](#).

### 3.2.2 Constructor & Destructor Documentation

#### 3.2.2.1 Node() [1/2]

```

template<class T >
Node< T >::Node

```

[Node](#) constructor function. Initializes the next and prev to nullptr.

Definition at line 17 of file [Node.cpp](#).

```

00017     {
00018         this->data = "";
00019         this->next = nullptr;
00020         this->prev = nullptr;
00021 }

```

#### 3.2.2.2 Node() [2/2]

```

template<class T >
Node< T >::Node (
    T data )

```

[Node](#) constructor function

## Parameters

<i>data</i>	- the data that the node will hold.
-------------	-------------------------------------

Definition at line 28 of file [Node.cpp](#).

```
00028     {
00029         this->data = data;
00030         this->next = nullptr;
00031         this->prev = nullptr;
00032     }
```

### 3.2.2.3 ~Node()

```
template<class T >
Node< T >::~~Node
```

[Node](#) destructor function. Resets the next and prev to nullptr.

Definition at line 39 of file [Node.cpp](#).

```
00039     {
00040         this->next = nullptr;
00041         this->prev = nullptr;
00042     }
```

## 3.2.3 Member Function Documentation

### 3.2.3.1 getData()

```
template<class T >
T Node< T >::getData
```

Definition at line 46 of file [Node.cpp](#).

```
00046 {return this->data;}
```

### 3.2.3.2 getNextNode()

```
template<class T >
Node< T > * Node< T >::getNextNode
```

Definition at line 49 of file [Node.cpp](#).

```
00049 {return this->next;}
```

### 3.2.3.3 getPrevNode()

```
template<class T >
Node< T > * Node< T >::getPrevNode
```

Definition at line 52 of file [Node.cpp](#).

```
00052 {return this->prev;}
```

#### 3.2.3.4 setData()

```
template<class T >
void Node< T >::setData (
    T data )
```

Definition at line 55 of file [Node.cpp](#).

```
00055 {this->data = data;}
```

#### 3.2.3.5 setNextNode()

```
template<class T >
void Node< T >::setNextNode (
    Node< T > * next )
```

Definition at line 58 of file [Node.cpp](#).

```
00058 {this->next = next;}
```

#### 3.2.3.6 setNextNodeNull()

```
template<class T >
void Node< T >::setNextNodeNull
```

Definition at line 64 of file [Node.cpp](#).

```
00064 {this->next = nullptr;}
```

#### 3.2.3.7 setPrevNode()

```
template<class T >
void Node< T >::setPrevNode (
    Node< T > * prev )
```

Definition at line 61 of file [Node.cpp](#).

```
00061 {this->prev = prev;}
```

#### 3.2.3.8 setPrevNodeNull()

```
template<class T >
void Node< T >::setPrevNodeNull
```

Definition at line 67 of file [Node.cpp](#).

```
00067 {this->prev = nullptr;}
```

The documentation for this class was generated from the following files:

- Node.hpp
- Node.cpp

## 3.3 Vault Class Reference

### Public Member Functions

- [Vault](#) (int startBal)
- bool [operator==](#) (const [Vault](#) &r)
- bool [operator!=](#) (const [Vault](#) &r)
- bool [operator<](#) (const [Vault](#) &r)
- bool [operator>](#) (const [Vault](#) &r)
- bool [operator<=](#) (const [Vault](#) &r)
- bool [operator>=](#) (const [Vault](#) &r)

### Friends

- std::ostream & [operator<<](#) (std::ostream &os, const [Vault](#) &v)

### 3.3.1 Detailed Description

Definition at line 11 of file [Vault.hpp](#).

### 3.3.2 Constructor & Destructor Documentation

#### 3.3.2.1 Vault() [1/2]

```
Vault::Vault ( )
```

Definition at line 12 of file [Vault.cpp](#).

```
00012     {  
00013         this->balance = 0;  
00014     }
```

#### 3.3.2.2 Vault() [2/2]

```
Vault::Vault (  
            int startBal )
```

Definition at line 16 of file [Vault.cpp](#).

```
00016     {  
00017         this->balance = startBal;  
00018     }
```

#### 3.3.2.3 ~Vault()

```
Vault::~Vault ( )
```

Definition at line 20 of file [Vault.cpp](#).

```
00020 {}
```

### 3.3.3 Member Function Documentation

#### 3.3.3.1 operator"!="()

```
bool Vault::operator!= (
    const Vault & r )
```

Definition at line 26 of file [Vault.cpp](#).

```
00026 {
00027     return this->balance != r.balance;
00028 }
```

#### 3.3.3.2 operator<()

```
bool Vault::operator< (
    const Vault & r )
```

Definition at line 30 of file [Vault.cpp](#).

```
00030 {
00031     return this->balance < r.balance;
00032 }
```

#### 3.3.3.3 operator<=()

```
bool Vault::operator<= (
    const Vault & r )
```

Definition at line 38 of file [Vault.cpp](#).

```
00038 {
00039     return this->balance <= r.balance;
00040 }
```

#### 3.3.3.4 operator==()

```
bool Vault::operator== (
    const Vault & r )
```

Definition at line 22 of file [Vault.cpp](#).

```
00022 {
00023     return this->balance == r.balance;
00024 }
```

#### 3.3.3.5 operator>()

```
bool Vault::operator> (
    const Vault & r )
```

Definition at line 34 of file [Vault.cpp](#).

```
00034 {
00035     return this->balance > r.balance;
00036 }
```

### 3.3.3.6 operator>=()

```
bool Vault::operator>= (
    const Vault & r )
```

Definition at line 42 of file [Vault.cpp](#).

```
00042     {
00043     return this->balance >= r.balance;
00044 };
```

## 3.3.4 Friends And Related Symbol Documentation

### 3.3.4.1 operator<<

```
std::ostream & operator<< (
    std::ostream & os,
    const Vault & v ) [friend]
```

Definition at line 26 of file [Vault.hpp](#).

```
00026     {
00027     os << "Vault with balance: " << v.balance;
00028     return os;
00029 }
```

The documentation for this class was generated from the following files:

- Vault.hpp
- Vault.cpp

# Chapter 4

## File Documentation

### 4.1 FunctionTests.hpp

```
00001 //
00002 // FunctionTests.hpp
00003 // CSC340GP
00004 //
00005 // Created by e on 5/13/23.
00006 //
00007
00008 #ifndef FunctionTests_hpp
00009 #define FunctionTests_hpp
00010
00011 #include "Vault.hpp"
00012
00013 void testAddRemove();
00014 void testSearch();
00015 void emptyMerge();
00016 void callingListEmptyMerge();
00017 void paramterListEmptyMerge();
00018 void twoNonEmptyMerge();
00019 void testMerge();
00020 void testMergeSort();
00021 void testAddFromFile();
00022 void testBubbleSort();
00023 void testInsert();
00024 void testBinarySearch();
00025 void testLinkedList();
00026 void demo();
00027 template<class T>
00028 bool assertion(T actual, T expected);
00029 template<class T>
00030 bool assertion(T actual, T expected, bool message);
00031
00035 void testAddRemove() {
00036     unsigned int i;
00037     LinkedList<int> intList = LinkedList<int>();
00038     LinkedList<std::string> stringList = LinkedList<std::string>();
00039
00040     for (i = 1; i <= 20; i++) {
00041         intList.add(i);
00042         stringList.add("Number " + std::to_string(i));
00043     }
00044
00045     assertion(intList.toString(), std::string("{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20}"));
00046     std::cout << "Added: ";
00047     std::cout << intList.toString() << std::endl;
00048
00049     assertion(stringList.toString(), std::string("{\"Number 1\", \"Number 2\", \"Number 3\", \"Number
4\", \"Number 5\", \"
        \"Number 6\", \"Number 7\", \"Number 8\", \"Number
9\", \"Number 10\", \"Number 11\", \"Number 12\", \"Number 13\", \"
        \"Number 14\", \"Number 15\", \"Number 16\",
17\", \"Number 18\", \"Number 19\", \"Number 20\"}"));
00052     std::cout << "Added: ";
00053     std::cout << stringList.toString() << std::endl;
00054
00055     for (i = 1; i <= 20; i += 2) {
00056         intList.remove(i);
00057         stringList.remove("Number " + std::to_string(i));
```

```

00058     }
00059
00060     assertion(intList.toString(), std::string("{2, 4, 6, 8, 10, 12, 14, 16, 18, 20}"));
00061     std::cout << "Removed: ";
00062     std::cout << intList.toString() << std::endl;
00063
00064     assertion(stringList.toString(), std::string("{\\\"Number 2\\\", \\\"Number 4\\\", \\\"Number 6\\\", \\\"Number
00065 8\\\", \\\"Number 10\\\", \"
                                \"\\\"Number 12\\\", \\\"Number 14\\\", \\\"Number 16\\\",
\\\"Number 18\\\", \\\"Number 20\\\"}"));
00066     std::cout << "Removed: ";
00067     std::cout << stringList.toString() << std::endl;
00068 }
00069
00073 void testSearch() {
00074     LinkedList<int> intListTest = LinkedList<int>();
00075     Node<int>* result;
00076
00077     std::cout << "[\\033[0;36m----\\033[0m] ";
00078     intListTest.print();
00079     result = intListTest.search(3);
00080     assertion(result, static_cast<Node<int>*>(nullptr));
00081     std::cout << "Empty Search -> ";
00082     std::cout << "Searching for 3: ";
00083     if (result) {
00084         std::cout << "Node was found" << std::endl;
00085     }
00086     else {
00087         std::cout << "Node not found" << std::endl;
00088     }
00089
00090     intListTest.add(1);
00091     std::cout << "[\\033[0;36m----\\033[0m] ";
00092     intListTest.print();
00093     result = intListTest.search(1);
00094     assertion(result->getData(), 1);
00095     std::cout << "One Item Search -> ";
00096     std::cout << "Searching for 1: ";
00097     if (result) {
00098         std::cout << "Node was found" << std::endl;
00099     }
00100     else {
00101         std::cout << "Node not found" << std::endl;
00102     }
00103
00104     std::cout << "[\\033[0;36m----\\033[0m] ";
00105     intListTest.print();
00106     result = intListTest.search(2);
00107     assertion(result, static_cast<Node<int>*>(nullptr));
00108     std::cout << "One Item Search -> ";
00109     std::cout << "Searching for 2: ";
00110     if (result) {
00111         std::cout << "Node was found" << std::endl;
00112     }
00113     else {
00114         std::cout << "Node not found" << std::endl;
00115     }
00116
00117     intListTest.add(10);
00118     intListTest.add(25);
00119     intListTest.add(30);
00120
00121     std::cout << "[\\033[0;36m----\\033[0m] ";
00122     intListTest.print();
00123     result = intListTest.search(30);
00124     assertion(result->getData(), 30);
00125     std::cout << "Multiple Item Search -> ";
00126     std::cout << "Searching for 30: ";
00127     if (result) {
00128         std::cout << "Node was found" << std::endl;
00129     }
00130     else {
00131         std::cout << "Node not found" << std::endl;
00132     }
00133
00134     std::cout << "[\\033[0;36m----\\033[0m] ";
00135     intListTest.print();
00136     result = intListTest.search(10000);
00137     assertion(result, static_cast<Node<int>*>(nullptr));
00138     std::cout << "Multiple Item Search -> ";
00139     std::cout << "Searching for 10000: ";
00140     if (result) {
00141         std::cout << "Node was found" << std::endl;
00142     }
00143     else {
00144         std::cout << "Node not found" << std::endl;
00145     }

```



```

00146 }
00147
00148
00152 void emptyMerge() {
00153     LinkedList<int> list1 = LinkedList<int>();
00154     LinkedList<int> list2 = LinkedList<int>();
00155     list1.mergeLists(&list2);
00156     std::string expect_output = "{}";
00157     std::string output = list1.toString();
00158     assertion(output, expect_output);
00159     std::cout << "Empty Merge Output: " << output << std::endl;
00160 }
00161
00165 void callingListEmptyMerge() {
00166     LinkedList<int> list1 = LinkedList<int>();
00167     LinkedList<int> list2 = LinkedList<int>();
00168     list2.add(25);
00169     list1.mergeLists(&list2);
00170     std::string expect_output = "{25}";
00171     std::string output = list1.toString();
00172     assertion(output, expect_output);
00173     std::cout << "Calling List Empty Output: " << output << std::endl;
00174 }
00175
00176
00180 void paramterListEmptyMerge() {
00181     LinkedList<int> list1 = LinkedList<int>();
00182     LinkedList<int> list2 = LinkedList<int>();
00183     list1.add(25);
00184     list1.mergeLists(&list2);
00185     std::string expect_output = "{25}";
00186     std::string output = list1.toString();
00187     assertion(output, expect_output);
00188     std::cout << "Parameter List Empty Output: " << output << std::endl;
00189 }
00190
00194 void twoNonEmptyMerge() {
00195     LinkedList<int> list1 = LinkedList<int>();
00196     LinkedList<int> list2 = LinkedList<int>();
00197     list1.add(65);
00198     list1.add(25);
00199     list1.add(35);
00200     list2.add(45);
00201     list2.add(90);
00202     list2.add(10);
00203     list1.mergeLists(&list2);
00204     std::string expect_output = "{10, 25, 35, 45, 65, 90}";
00205     std::string output = list1.toString();
00206     assertion(output, expect_output);
00207     std::cout << "Two Non Empty List Output: " << output << std::endl;
00208 }
00209
00213 void testMerge() {
00214     emptyMerge();
00215     callingListEmptyMerge();
00216     paramterListEmptyMerge();
00217     twoNonEmptyMerge();
00218 }
00219
00223 void testMergeSort() {
00224     std::string stringItems[26] = {
00225         "Quebec", "Victor", "November", "Mike", "Charlie", "X-Ray",
00226         "Zulu", "Yankee", "Juliett", "Uniform", "Oscar", "Lima", "Romeo",
00227         "Bravo", "Tango", "Kilo", "Foxtrot", "India", "Delta", "Sierra",
00228         "Golf", "Alpha", "Papa", "Echo", "Hotel", "Whiskey"
00229     };
00230     LinkedList<std::string> stringList = LinkedList<std::string>();
00231
00232     for (unsigned int i = 0; i < 26; ++i) {
00233         stringList.add(stringItems[i]);
00234     }
00235
00236     int intItems[26] = {23, 1, 21, 5, 4, 17, 15, 13, 3, 2, 12, 19, 6, 10, 20, 26, 18, 9, 25, 24, 16,
14, 11, 22, 8, 7};
00237     LinkedList<int> intList = LinkedList<int>();
00238
00239     for (unsigned int i = 0; i < 26; ++i) {
00240         intList.add(intItems[i]);
00241     }
00242
00243     std::cout << "[033[0;36m---\033[0m] ";
00244     std::cout << "Before Sort: ";
00245     stringList.print();
00246
00247     stringList.mergeSort();
00248
00249     assertion(stringList.toString(), std::string("{\\Alpha\\", "\\Bravo\\", "\\Charlie\\", "\\Delta\\",

```

```

    \ "Echo", \ "Foxtrot", \ "Golf", "
00250     \ "Hotel", \ "India", \ "Juliett", \ "Kilo",
    \ "Lima", \ "Mike", \ "November", \ "Oscar", \ "Papa", \ "Quebec", \ "Romeo", "
00251     \ "Sierra", \ "Tango", \ "Uniform", \ "Victor",
    \ "Whiskey", \ "X-Ray", \ "Yankee", \ "Zulu"}));
00252     std::cout << "After Sort: ";
00253     stringList.print();
00254
00255     std::cout << std::endl;
00256
00257     std::cout << "[\033[0;36m---\033[0m] ";
00258     std::cout << "Before Sort: ";
00259     intList.print();
00260
00261     intList.mergeSort();
00262
00263     assertion(intList.toString(), std::string("{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26}"));
00264     std::cout << "After Sort: ";
00265     intList.print();
00266 }
00267
00268
00269 void testAddFromFile() {
00270     std::string fileName = "file333.txt";
00271     LinkedList<std::string>* list = new LinkedList<std::string>;
00272     Node<std::string>* node = new Node<std::string>;
00273
00274     list->add("The");
00275     list->add("best");
00276     list->add("team");
00277     list->add("in");
00278     list->add("CSC340");
00279
00280     std::cout << "Before adding data from file: " << list->toString() << std::endl;
00281
00282     list->addFromFile(fileName);
00283
00284     std::cout << "After adding data from file: " << list->toString() << std::endl;
00285
00286     node = list->search("Dummy");
00287
00288     if (node->getNextNode()->getData() == "Text") {
00289         std::cout << "Test passed" << std::endl;
00290     }
00291     else {
00292         std::cerr << "Test failed" << std::endl;
00293     }
00294 }
00295
00296 std::string read_file() {
00297     std::string data = "";
00298     std::ifstream file;
00299     std::string line = "";
00300     file.open("file333.txt");
00301
00302     if (!file.is_open()) {
00303         std::cerr << "Error opening file" << std::endl;
00304     }
00305
00306     while (file >> line) {
00307         data += line;
00308     }
00309
00310     file.close();
00311
00312     return data;
00313 }
00314
00315 void testBubbleSort() {
00316     LinkedList<int> intList;
00317     LinkedList<std::string> stringList;
00318
00319     int intItems[10] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
00320     for (unsigned int i = 0; i < 10; ++i) {
00321         intList.add(intItems[i]);
00322     }
00323
00324     std::cout << "[\033[0;36m---\033[0m] ";
00325     std::cout << "Before Sorting: " << intList.toString() << std::endl;
00326     intList.bubbleSort();
00327     assertion(intList.toString(), std::string("{1, 1, 2, 3, 3, 4, 5, 5, 6, 9}"));
00328     std::cout << "After Sorting: " << intList.toString() << std::endl;
00329
00330     std::string stringItems[4] = {"zzz", "bbb", "eee", "ddd"};
00331     for (unsigned int i = 0; i < 4; ++i) {
00332         stringList.add(stringItems[i]);
00333     }

```

```

00336     }
00337
00338     std::cout << "[\033[0;36m---\033[0m] ";
00339     std::cout << "Before Sorting: " << stringList.toString() << std::endl;
00340     stringList.bubbleSort();
00341     assertion(stringList.toString(), std::string("{\"bbb\", \"ddd\", \"eee\", \"zzz\"}"));
00342     std::cout << "After Sorting: " << stringList.toString() << std::endl;
00343 }
00344
00352 template<class T>
00353 bool assertion(T actual, T expected) {
00354     if (actual == expected) {
00355         std::cout << "[\033[0;32mPass\033[0m] ";
00356         return true;
00357     }
00358     else {
00359         std::cout << "[\033[0;31mFail\033[0m] ";
00360         std::cout << "Expected: " << expected << std::endl;
00361         std::cout << "[\033[0;31m--->\033[0m] ";
00362         std::cout << "Actual: " << actual << std::endl;
00363         std::cout << "[\033[0;36m---\033[0m] ";
00364         return false;
00365     }
00366 }
00367
00368 template<class T>
00369 bool assertion(T actual, T expected, bool message) {
00370     if (message) {
00371         return assertion(actual, expected);
00372     }
00373     else {
00374         return actual == expected;
00375     }
00376 }
00377
00381 void testInsert() {
00382     LinkedList<int> intList;
00383     LinkedList<std::string> stringList;
00384
00385     // pi
00386     int intItems[9] = {3, 1, 4, 1, 5, 9, 2, 6, 5};
00387     for (unsigned int i = 0; i < 9; ++i) {
00388         intList.insert(intItems[i]);
00389         std::cout << "[\033[0;36m---\033[0m] ";
00390         intList.print();
00391     }
00392
00393     intList.insert(3);
00394     assertion(intList.toString(), std::string("{1, 1, 2, 3, 3, 4, 5, 5, 6, 9}"));
00395     intList.print();
00396
00397     std::string stringItems[2] = {"pug", "bear"};
00398     for (unsigned int i = 0; i < 2; ++i) {
00399         stringList.insert(stringItems[i]);
00400         std::cout << "[\033[0;36m---\033[0m] ";
00401         stringList.print();
00402     }
00403
00404     stringList.insert("zebra");
00405     assertion(stringList.toString(), std::string("{\"bear\", \"pug\", \"zebra\"}"));
00406     stringList.print();
00407 }
00408
00412 void testBinarySearch() {
00413     LinkedList<int> listForSearch = LinkedList<int>();
00414     Node<int>* result;
00415
00416     std::cout << "[\033[0;36m---\033[0m] ";
00417     listForSearch.print();
00418     result = listForSearch.binarySearch(2);
00419     assertion(result, static_cast<Node<int>*>(nullptr));
00420     std::cout << "Empty Binary Search -> ";
00421     std::cout << "Searching for 2: ";
00422     if (result) {
00423         std::cout << "Node was found" << std::endl;
00424     }
00425     else {
00426         std::cout << "Node not found" << std::endl;
00427     }
00428
00429     listForSearch.add(2);
00430     std::cout << "[\033[0;36m---\033[0m] ";
00431     listForSearch.print();
00432     result = listForSearch.binarySearch(2);
00433     assertion(result->getData(), 2);
00434     std::cout << "One Item Binary Search -> ";
00435     std::cout << "Searching for 2: ";

```

```

00436     if (result) {
00437         std::cout << "Node was found" << std::endl;
00438     }
00439     else {
00440         std::cout << "Node not found" << std::endl;
00441     }
00442
00443     listForSearch.insert(1);
00444     std::cout << "[\033[0;36m---\033[0m] ";
00445     listForSearch.print();
00446     result = listForSearch.binarySearch(1);
00447     assertion(result->getData(), 1);
00448     std::cout << "Two Item List Binary Search - First Node -> ";
00449     std::cout << "Searching for 1: ";
00450     if (result) {
00451         std::cout << "Node was found" << std::endl;
00452     }
00453     else {
00454         std::cout << "Node not found" << std::endl;
00455     }
00456
00457     listForSearch.insert(10);
00458     std::cout << "[\033[0;36m---\033[0m] ";
00459     listForSearch.print();
00460     result = listForSearch.binarySearch(10);
00461     assertion(result->getData(), 10);
00462     std::cout << "Tree Item List Binary Search - Last Node -> ";
00463     std::cout << "Searching for 10: ";
00464     if (result) {
00465         std::cout << "Node was found with a " << result->getData() << std::endl;
00466     }
00467     else {
00468         std::cout << "Node not found" << std::endl;
00469     }
00470 }
00471
00475 void testLinkedList() {
00476     std::cout << " -- Add and Remove Node Test -- " << std::endl;
00477     testAddRemove();
00478     std::cout << std::endl;
00479
00480     std::cout << " -- Insert Node Test -- " << std::endl;
00481     testInsert();
00482     std::cout << std::endl;
00483
00484     std::cout << " -- Search For Node Test -- " << std::endl;
00485     testSearch();
00486     std::cout << std::endl;
00487
00488     std::cout << " -- Binary Search Test -- " << std::endl;
00489     testBinarySearch();
00490     std::cout << std::endl;
00491
00492     std::cout << " -- Merge LinkedLists Test -- " << std::endl;
00493     testMerge();
00494     std::cout << std::endl;
00495
00496     std::cout << " -- Bubble Sort Test -- " << std::endl;
00497     testBubbleSort();
00498     std::cout << std::endl;
00499
00500     std::cout << " -- Merge Sort Test -- " << std::endl;
00501     testMergeSort();
00502     std::cout << std::endl;
00503 }
00504
00505 void demo() {
00506
00507     LinkedList<Vault> BankSystem = LinkedList<Vault>();
00508     LinkedList<Vault> BankSystem2 = LinkedList<Vault>();
00509     BankSystem2.add(Vault(100));
00510     BankSystem2.add(Vault(100000));
00511     BankSystem2.add(Vault(13));
00512     int choice = 0;
00513     while (choice != -1) {
00514         std::cout << "Banking System Main Menu: " << std::endl;
00515         std::cout << "Enter '1' to print the current Bank System" << std::endl;
00516         std::cout << "Enter '2' to add a vault to the current system" << std::endl;
00517         std::cout << "Enter '3' to search for a vault" << std::endl;
00518         std::cout << "Enter '4' to do a binary search must be sorted" << std::endl;
00519         std::cout << "Enter '5' to bubble sort the Bank System" << std::endl;
00520         std::cout << "Enter '6' to merge sort the Bank System" << std::endl;
00521         std::cout << "Enter '7' to merge another Bank System into this one" << std::endl;
00522         std::cout << "Enter '-1' to exit the management system" << std::endl;
00523         std::cin >> choice;
00524         switch (choice) {
00525             case 1:

```

```

00526         std::cout << "Current Bank System: ";
00527         BankSystem.print();
00528         std::cout << std::endl;
00529         break;
00530     case 2:
00531         int toAdd;
00532         std::cout << "Enter the value of the new vault to add to the system: ";
00533         std::cin >> toAdd;
00534         BankSystem.add(Vault(toAdd));
00535         std::cout << "Added" << std::endl;
00536         break;
00537     case 3:
00538         int searchTarget;
00539         std::cout << "Enter the value of the target vault ";
00540         std::cin >> searchTarget;
00541         if (BankSystem.search(searchTarget)) {
00542             std::cout << "Vault located" << std::endl;
00543         }
00544         else {
00545             std::cout << "Vault not located" << std::endl;
00546         }
00547         break;
00548     case 4:
00549         int binaryTarget;
00550         std::cout << "Enter the value of the target vault ";
00551         std::cin >> binaryTarget;
00552         if (BankSystem.binarySearch(binaryTarget)) {
00553             std::cout << "Vault located" << std::endl;
00554         }
00555         else {
00556             std::cout << "Vault not located" << std::endl;
00557         }
00558         break;
00559     case 5:
00560         std::cout << "Bubble Sorting" << std::endl;
00561         BankSystem.bubbleSort();
00562         std::cout << "Sorted!" << std::endl;
00563         break;
00564     case 6:
00565         std::cout << "Merge Sorting" << std::endl;
00566         BankSystem.mergeSort();
00567         std::cout << "Sorted!" << std::endl;
00568         break;
00569     case 7:
00570         std::cout << "Merging other Bank System" << std::endl;
00571         std::cout << "Other system: ";
00572         BankSystem2.print();
00573         BankSystem.mergeLists(&BankSystem2);
00574         std::cout << "New Merge System: ";
00575         BankSystem.print();
00576         break;
00577     default:
00578         std::cout << "Enter a value between 1 and 6 or -1";
00579 }
00580 }
00581 }
00582 }
00583 }
00584 }
00585 #endif /* FunctionTests_hpp */

```

## 4.2 LinkedList.cpp

```

00001
00007 #ifndef LINKEDLIST_CPP
00008 #define LINKEDLIST_CPP
00009 #include <fstream>
00010 #include "LinkedList.hpp"
00011
00012 template<class T>
00013 LinkedList<T>::LinkedList() {
00014     this->head = nullptr;
00015     this->tail = nullptr;
00016 }
00017
00018 template<class T>
00019 LinkedList<T>::~~LinkedList() {
00020     clear();
00021 }
00022
00023 template<class T>
00024 void LinkedList<T>::clear() {
00025     Node<T>* nodeToDelete = head;

```

```

00026     while (head != nullptr) {
00027         head = head->getNextNode();
00028         delete nodeToDelete;
00029         nodeToDelete = head;
00030     }
00031 }
00032
00033 template<class T>
00034 void LinkedList<T>::insert(T data) {
00035     Node<T>* newNode = new Node<T>(data);
00036     if (this->head == nullptr) {
00037         this->head = newNode;
00038         this->tail = newNode;
00039         return;
00040     }
00041     if (data <= this->head->getData()) {
00042         newNode->setNextNode(head);
00043         this->head->setPrevNode(newNode);
00044         this->head = newNode;
00045         return;
00046     }
00047     if (data >= tail->getData()) {
00048         newNode->setPrevNode(tail);
00049         this->tail->setNextNode(newNode);
00050         this->tail = newNode;
00051         return;
00052     }
00053     Node<T>* temp = this->head;
00054     while (temp->getNextNode() != nullptr && temp->getNextNode()->getData() < data) {
00055         temp = temp->getNextNode();
00056     }
00057 }
00058
00059 newNode->setNextNode(temp->getNextNode());
00060 temp->setNextNode(newNode);
00061 newNode->setPrevNode(temp);
00062 newNode->getNextNode()->setPrevNode(newNode);
00063 }
00064
00065 template<class T>
00066 void LinkedList<T>::add(T data) {
00067     Node<T>* newNode = new Node<T>(data);
00068
00069     if (this->head == nullptr) {
00070         this->head = newNode;
00071         this->tail = newNode;
00072     }
00073     else {
00074         this->tail->setNextNode(newNode);
00075         newNode->setPrevNode(tail);
00076         this->tail = newNode;
00077     }
00078 }
00079
00080 template<class T>
00081 void LinkedList<T>::remove(T data) {
00082     Node<T>* nodeToDelete = search(data);
00083     if (nodeToDelete != nullptr) {
00084         if (nodeToDelete == this->head) {
00085             this->head = head->getNextNode();
00086             this->head->setPrevNodeNull();
00087         }
00088         else if (nodeToDelete == this->tail) {
00089             this->tail = tail->getPrevNode();
00090             this->tail->setNextNodeNull();
00091         }
00092         else {
00093             Node<T>* prevNode = nodeToDelete->getPrevNode();
00094             Node<T>* nextNode = nodeToDelete->getNextNode();
00095             prevNode->setNextNode(nextNode);
00096             nextNode->setPrevNode(prevNode);
00097         }
00098         delete nodeToDelete;
00099     }
00100 }
00101
00102 template<class T>
00103 Node<T>* LinkedList<T>::search(T data) {
00104     Node<T>* temp = this->head;
00105
00106     while (temp) {
00107         if (temp->getData() == data) {
00108             return temp;
00109         }
00110
00111         temp = temp->getNextNode();
00112     }

```

```

00113
00114     return nullptr;
00115 }
00116
00121 template<class T>
00122 std::string LinkedList<T>::toString() {
00123     Node<T>* temp = this->head;
00124     std::string output = "";
00125     std::string quote = (typeid(T).name() == typeid(std::string("")).name()) ? "\"" : "";
00126
00127     while (temp != nullptr) {
00128         T val = temp->getData();
00129         output += quote + to_string(val) + quote;
00130         temp = temp->getNextNode();
00131         if (temp != nullptr) {
00132             output += ", ";
00133         }
00134     }
00135     return "{" + output + "}";
00136 }
00137
00141 template<class T>
00142 void LinkedList<T>::mergeSort() {
00143     // base case: 1 or 0 Nodes
00144     if (this->head == nullptr || this->head->getNextNode() == nullptr) {
00145         return;
00146     }
00147
00148     // split the LinkedList in half
00149     Node<T>* subHead1 = this->head;
00150     Node<T>* subHead2 = findMid(this->head, this->tail);
00151     Node<T>* subTail1 = subHead2->getPrevNode();
00152     Node<T>* subTail2 = this->tail;
00153
00154     // detach the two halves
00155     subTail1->setNextNodeNull();
00156     subHead2->setPrevNodeNull();
00157
00158     // recurse first half
00159     this->head = subHead1;
00160     this->tail = subTail1;
00161     mergeSort();
00162     subHead1 = this->head;
00163     subTail1 = this->tail;
00164
00165     // recurse second half
00166     this->head = subHead2;
00167     this->tail = subTail2;
00168     mergeSort();
00169     subHead2 = this->head;
00170     subTail2 = this->tail;
00171
00172     // merge both halves
00173     this->head = nullptr;
00174     Node<T>* nodeptr = nullptr;
00175
00176     // compare head of both halves
00177     while (subHead1 != nullptr && subHead2 != nullptr) {
00178         if (subHead1->getData() < subHead2->getData()) {
00179             if (this->head == nullptr) {
00180                 this->head = subHead1;
00181                 nodeptr = subHead1;
00182             }
00183             else {
00184                 nodeptr->setNextNode(subHead1);
00185                 subHead1->setPrevNode(nodeptr);
00186                 nodeptr = subHead1;
00187             }
00188             subHead1 = subHead1->getNextNode();
00189         }
00190         else {
00191             if (this->head == nullptr) {
00192                 this->head = subHead2;
00193                 nodeptr = subHead2;
00194             }
00195             else {
00196                 nodeptr->setNextNode(subHead2);
00197                 subHead1->setPrevNode(nodeptr);
00198                 nodeptr = subHead2;
00199             }
00200             subHead2 = subHead2->getNextNode();
00201         }
00202     }
00203
00204     // add the rest of first half to the main LinkedList
00205     while (subHead1 != nullptr) {
00206         if (this->head == nullptr) {

```

```

00207         this->head = subHead1;
00208         nodeptr = subHead1;
00209     }
00210     else {
00211         nodeptr->setNextNode(subHead1);
00212         subHead1->setPrevNode(nodeptr);
00213         nodeptr = subHead1;
00214     }
00215     subHead1 = subHead1->getNextNode();
00216 }
00217
00218 // add the rest of second half to the main LinkedList
00219 while (subHead2 != nullptr) {
00220     if (this->head == nullptr) {
00221         this->head = subHead2;
00222         nodeptr = subHead2;
00223     }
00224     else {
00225         nodeptr->setNextNode(subHead2);
00226         subHead2->setPrevNode(nodeptr);
00227         nodeptr = subHead2;
00228     }
00229     subHead2 = subHead2->getNextNode();
00230 }
00231
00232 this->tail = nodeptr;
00233 }
00234
00235 template<class T>
00236 void LinkedList<T>::bubbleSort() {
00237     // do not sort if empty or one
00238     if (this->head == nullptr || this->head->getNextNode() == nullptr)
00239         return;
00240
00241     bool swap;
00242     Node<T>* current = this->head;
00243     Node<T>* sorttail = nullptr;
00244
00245     while (current != sorttail) {
00246         swap = false;
00247         Node<T>* current2 = this->head;
00248
00249         while (current2->getNextNode() != sorttail) {
00250             if (current2->getData() > current2->getNextNode()->getData()) {
00251                 T temp = current2->getData();
00252                 current2->setData(current2->getNextNode()->getData());
00253                 current2->getNextNode()->setData(temp);
00254                 swap = true;
00255             }
00256             current2 = current2->getNextNode();
00257         }
00258         sorttail = current2; // update tail to last swap
00259         if (!swap)
00260             break; // if no swap the list is already sorted
00261     }
00262 }
00263
00264 template<class T>
00265 void LinkedList<T>::addFromFile (std::string fileName) {
00266     std::ifstream file;
00267     T data;
00268
00269     file.open(fileName);
00270
00271     if (file.peek() == std::ifstream::traits_type::eof()) {
00272         std::cerr << "Error: File is empty" << std::endl;
00273         exit(1);
00274     }
00275
00276     if (!file.is_open()) {
00277         std::cerr << "Error opening file" << std::endl;
00278         exit(1);
00279     }
00280
00281     while (file >> data) {
00282         if (file.eof()) {
00283             break;
00284         }
00285         this->add(data);
00286     }
00287
00288     if (!file.eof()) {
00289         std::cerr << "Error reaching end of file" << std::endl;
00290         exit(1);
00291     }
00292 }
00293
00294
00295
00296

```



```

00297     file.close();
00298 }
00299
00306 template<class T>
00307 void LinkedList<T>::mergeLists(const LinkedList<T>* listTwo) {
00308     if (!this->head && !listTwo->head) {
00309         return;
00310     }
00311     else {
00312         mergeSort();
00313         Node<T>* temp = listTwo->head;
00314         while (temp != nullptr) {
00315             this->insert(temp->getData());
00316             temp = temp->getNextNode();
00317         }
00318     }
00319 }
00320
00321 template<class T>
00322 void LinkedList<T>::print() {
00323     Node<T>* temp = this->head;
00324
00325     if (this->head == nullptr) {
00326         std::cout << "The linked list is empty." << std::endl;
00327         return;
00328     }
00329
00330     while (temp != nullptr) {
00331         std::cout << temp->getData() << " ";
00332         temp = temp->getNextNode();
00333     }
00334
00335     std::cout << std::endl;
00336 }
00337
00344 template<class T>
00345 Node<T>* LinkedList<T>::binarySearch(T target) {
00346     Node<T>* searchHead = this->head;
00347     Node<T>* searchTail = this->tail;
00348     Node<T>* searchMid = findMid(searchHead, searchTail);
00349     if (searchHead) {
00350         while (searchHead->getData() <= searchTail->getData()) {
00351             if (target == searchHead->getData()) {
00352                 return searchHead;
00353             }
00354             else if (target == searchMid->getData()) {
00355                 return searchMid;
00356             }
00357             else if (target == searchTail->getData()) {
00358                 return searchTail;
00359             }
00360
00361             if (target < searchMid->getData()) {
00362                 searchHead = searchHead->getNextNode();
00363                 searchTail = searchMid->getPrevNode();
00364                 searchMid = findMid(searchHead, searchTail);
00365             }
00366             else if (target < searchMid->getData()) {
00367                 searchHead = searchMid->getNextNode();
00368                 searchTail = searchTail->getPrevNode();
00369                 searchMid = findMid(searchHead, searchTail);
00370             }
00371         }
00372     }
00373
00374     return nullptr;
00375 }
00376
00377 #endif

```

## 4.3 LinkedList.hpp

```

00001 //
00002 //  LinkedList.hpp
00003 //  CSC340GP
00004 //
00005 //  Created by e on 5/13/23.
00006 //
00007
00008 #ifndef LINKEDLIST_HPP
00009 #define LINKEDLIST_HPP
00010 #include "Node.hpp"
00011 #include <iostream>

```

```

00012 #include <string>
00013 #include <sstream>
00014
00015 template<class T>
00016 class LinkedList {
00017 public:
00020     LinkedList();
00022     ~LinkedList();
00024     void clear();
00027     void insert(T data);
00030     void add(T data);
00033     void remove(T data);
00037     Node<T>* search(T data);
00038     std::string toString();
00039     void mergeSort();
00040     void bubbleSort();
00043     void addFromFile(std::string fileName);
00044     void mergeLists(const LinkedList<T>* listTwo);
00046     void print();
00047     Node<T>* binarySearch(T target);
00048
00049 private:
00050     Node<T>* head;
00051     Node<T>* tail;
00052
00057     std::string to_string(const T& obj) {
00058         std::ostringstream oss{};
00059         oss << obj;
00060         return oss.str();
00061     }
00062
00071     static Node<T>* findMid(Node<T>* start, Node<T>* end) {
00072         bool flip = true;
00073         while (start != end) {
00074             if (flip) {
00075                 start = start->getNextNode();
00076             }
00077             else {
00078                 end = end->getPrevNode();
00079             }
00080             flip = flip ? false : true;
00081         }
00082         return start;
00083     }
00084 };
00085
00086 #include "LinkedList.cpp"
00087 #endif /* LinkedList_hpp */

```

## 4.4 main.cpp

```

00001 /*****
00015 #include "LinkedList.hpp"
00016 #include "FunctionTests.hpp"
00017
00018 int main(int argc, const char* argv[]) {
00019
00020     testLinkedList();
00021     demo();
00022
00023     return 0;
00024 }

```

## 4.5 Node.cpp

```

00001
00005 #ifndef NODE_CPP
00006 #define NODE_CPP
00007
00008 #include "Node.hpp"
00009
00016 template<class T>
00017 Node<T>::Node() {
00018     this->data = "";
00019     this->next = nullptr;
00020     this->prev = nullptr;
00021 }
00022
00027 template<class T>

```

```

00028 Node<T>::Node(T data) {
00029     this->data = data;
00030     this->next = nullptr;
00031     this->prev = nullptr;
00032 }
00033
00038 template<class T>
00039 Node<T>::~Node() {
00040     this->next = nullptr;
00041     this->prev = nullptr;
00042 }
00043
00044 // getters
00045 template<class T>
00046 T Node<T>::getData() {return this->data;}
00047
00048 template<class T>
00049 Node<T>* Node<T>::getNextNode() {return this->next;}
00050
00051 template<class T>
00052 Node<T>* Node<T>::getPrevNode() {return this->prev;}
00053
00054 template<class T>
00055 void Node<T>::setData(T data) {this->data = data;}
00056
00057 template<class T>
00058 void Node<T>::setNextNode(Node* next) {this->next = next;}
00059
00060 template<class T>
00061 void Node<T>::setPrevNode(Node* prev) {this->prev = prev;}
00062
00063 template<class T>
00064 void Node<T>::setNextNodeNull() {this->next = nullptr;}
00065
00066 template<class T>
00067 void Node<T>::setPrevNodeNull() {this->prev = nullptr;}
00068 #endif

```

## 4.6 Node.hpp

```

00001 //
00002 // Node.hpp
00003 // CSC340GP
00004 //
00005 // Created by e on 5/13/23.
00006 //
00007
00008 #ifndef NODE_HPP
00009 #define NODE_HPP
00010
00011 template<class T>
00012 class Node {
00013 public:
00014     Node();
00015     Node(T data);
00016     ~Node();
00017     T getData();
00018
00019     Node* getNextNode();
00020     Node* getPrevNode();
00021
00022     void setData(T data);
00023     void setNextNode(Node* next);
00024     void setPrevNode(Node* prev);
00025     void setNextNodeNull();
00026     void setPrevNodeNull();
00027
00028 private:
00029     T data;
00030     Node<T>* next;
00031     Node<T>* prev;
00032 };
00033 #include "Node.cpp"
00034 #endif /* Node_hpp */

```

## 4.7 Vault.cpp

```

00001 //
00002 // Vault.cpp

```

```

00003 // CSC340GP
00004 //
00005 // Created by e on 5/18/23.
00006 //
00007 #ifndef Vault_cpp
00008 #define Vault_cpp
00009
00010 #include "Vault.hpp"
00011
00012 Vault::Vault() {
00013     this->balance = 0;
00014 }
00015
00016 Vault::Vault(int startBal) {
00017     this->balance = startBal;
00018 }
00019
00020 Vault::~Vault() {}
00021
00022 bool Vault::operator==(const Vault& r) {
00023     return this->balance == r.balance;
00024 }
00025
00026 bool Vault::operator!=(const Vault& r) {
00027     return this->balance != r.balance;
00028 }
00029
00030 bool Vault::operator<(const Vault& r) {
00031     return this->balance < r.balance;
00032 }
00033
00034 bool Vault::operator>(const Vault& r) {
00035     return this->balance > r.balance;
00036 }
00037
00038 bool Vault::operator<=(const Vault& r) {
00039     return this->balance <= r.balance;
00040 }
00041
00042 bool Vault::operator>=(const Vault& r) {
00043     return this->balance >= r.balance;
00044 };
00045 #endif

```

## 4.8 Vault.hpp

```

00001 //
00002 // Vault.hpp
00003 // CSC340GP
00004 //
00005 // Created by e on 5/18/23.
00006 //
00007
00008 #ifndef Vault_hpp
00009 #define Vault_hpp
00010
00011 class Vault {
00012 public:
00013     Vault();
00014     Vault(int startBal);
00015     ~Vault();
00016
00017     bool operator==(const Vault& r);
00018     bool operator!=(const Vault& r);
00019     bool operator<(const Vault& r);
00020     bool operator>(const Vault& r);
00021     bool operator<=(const Vault& r);
00022     bool operator>=(const Vault& r);
00023
00024 private:
00025     int balance;
00026     friend std::ostream& operator<<(std::ostream& os, const Vault& v) {
00027         os << "Vault with balance: " << v.balance;
00028         return os;
00029     }
00030 };
00031 #include "Vault.cpp"
00032 #endif /* Vault_hpp */

```

# Index

- ~LinkedList
  - LinkedList< T >, 6
- ~Node
  - Node< T >, 14
- ~Vault
  - Vault, 16
- add
  - LinkedList< T >, 6
- addFromFile
  - LinkedList< T >, 6
- binarySearch
  - LinkedList< T >, 7
- bubbleSort
  - LinkedList< T >, 8
- clear
  - LinkedList< T >, 8
- getData
  - Node< T >, 14
- getNextNode
  - Node< T >, 14
- getPrevNode
  - Node< T >, 14
- insert
  - LinkedList< T >, 8
- LinkedList
  - LinkedList< T >, 6
- LinkedList< T >, 5
  - ~LinkedList, 6
  - add, 6
  - addFromFile, 6
  - binarySearch, 7
  - bubbleSort, 8
  - clear, 8
  - insert, 8
  - LinkedList, 6
  - mergeLists, 9
  - mergeSort, 10
  - print, 11
  - remove, 11
  - search, 12
  - toString, 12
- mergeLists
  - LinkedList< T >, 9
- mergeSort
  - LinkedList< T >, 10
- Node
  - Node< T >, 13
- Node< T >, 13
  - ~Node, 14
  - getData, 14
  - getNextNode, 14
  - getPrevNode, 14
  - Node, 13
  - setData, 14
  - setNextNode, 15
  - setNextNodeNull, 15
  - setPrevNode, 15
  - setPrevNodeNull, 15
- operator!=
  - Vault, 17
- operator<
  - Vault, 17
- operator<<
  - Vault, 18
- operator<=
  - Vault, 17
- operator>
  - Vault, 17
- operator>=
  - Vault, 17
- operator==
  - Vault, 17
- print
  - LinkedList< T >, 11
- remove
  - LinkedList< T >, 11
- search
  - LinkedList< T >, 12
- setData
  - Node< T >, 14
- setNextNode
  - Node< T >, 15
- setNextNodeNull
  - Node< T >, 15
- setPrevNode
  - Node< T >, 15
- setPrevNodeNull
  - Node< T >, 15
- toString
  - LinkedList< T >, 10

LinkedList< T >, [12](#)

Vault, [16](#)

~Vault, [16](#)

operator!=, [17](#)

operator<, [17](#)

operator<<, [18](#)

operator<=, [17](#)

operator>, [17](#)

operator>=, [17](#)

operator==, [17](#)

Vault, [16](#)