A Generic Linked List

Generated by Doxygen 1.9.7

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

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

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 byLinked 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

data	- the data a Node 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

fileName	- 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

target	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 byLinked List.

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

data	- the data a Node 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

listTwo	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

data	- the data a Node 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

data	- the data a Node 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 
00106     while (temp) {
00107         if (temp->getData() == data) {
00108             return temp;
00109         }
00110 
00111         temp = temp->getNextNode();
00112     }
00113 
00114     return nullptr;
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 
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 }
```

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

data	- 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 deconstructor 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

```
//
//  FunctionTests.hpp
//  CSC340GP
//
//  Created by e on 5/13/23.
//

#ifndef FunctionTests_hpp
#define FunctionTests_hpp

#include "Vault.hpp"

void testAddRemove();
void testSearch();
void emptyMerge();
void callingListEmptyMerge();
void paramterListEmptyMerge();
void twoNonEmptyMerge();
void testMerge();
void testMergeSort();
void testAddFromFile();
void testBubbleSort();
void testInsert();
void testBinarySearch();
void testLinkedList();
void demo();
template<class T>
bool assertion(T actual, T expected);
template<class T>
bool assertion(T actual, T expected, bool message);

void testAddRemove() {
    unsigned int i;
    LinkedList<int> intList = LinkedList<int>();
    LinkedList<std::string> stringList = LinkedList<std::string>();

    for (i = 1; i <= 20; i++) {
        intList.add(i);
        stringList.add("Number " + std::to_string(i));
    }

    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}"));
    std::cout « "Added: ";
    std::cout « intList.toString() « std::endl;

    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\", \"Number 17\", \"Number 18\", \"Number 19\", \"Number 20\"}"));
    std::cout « "Added: ";
    std::cout « stringList.toString() « std::endl;

    for (i = 1; i <= 20; i += 2) {
        intList.remove(i);
        stringList.remove("Number " + std::to_string(i));
    }

    assertion(intList.toString(), std::string("{2, 4, 6, 8, 10, 12, 14, 16, 18, 20}"));
    std::cout « "Removed: ";
    std::cout « intList.toString() « std::endl;

    assertion(stringList.toString(), std::string("{\"Number 2\", \"Number 4\", \"Number 6\", \"Number 8\", \"Number 10\", "
                                                 "\"Number 12\", \"Number 14\", \"Number 16\", \"Number 18\", \"Number 20\"}"));
    std::cout « "Removed: ";
    std::cout « stringList.toString() « std::endl;
}

void testSearch() {
    LinkedList<int> intListTest = LinkedList<int>();
    Node<int>* result;

    std::cout « "[\033[0;36m----\033[0m] ";
    intListTest.print();
    result = intListTest.search(3);
    assertion(result, static_cast<Node<int>*>(nullptr));
    std::cout « "Empty Search -> ";
    std::cout « "Searching for 3: ";
    if (result) {
        std::cout « "Node was found" « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }

    intListTest.add(1);
    std::cout « "[\033[0;36m----\033[0m] ";
    intListTest.print();
    result = intListTest.search(1);
    assertion(result->getData(), 1);
    std::cout « "One Item Search -> ";
    std::cout « "Searching for 1: ";
    if (result) {
        std::cout « "Node was found" « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }

    std::cout « "[\033[0;36m----\033[0m] ";
    intListTest.print();
    result = intListTest.search(2);
    assertion(result, static_cast<Node<int>*>(nullptr));
    std::cout « "One Item Search -> ";
    std::cout « "Searching for 2: ";
    if (result) {
        std::cout « "Node was found" « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }

    intListTest.add(10);
    intListTest.add(25);
    intListTest.add(30);

    std::cout « "[\033[0;36m----\033[0m] ";
    intListTest.print();
    result = intListTest.search(30);
    assertion(result->getData(), 30);
    std::cout « "Multiple Item Search -> ";
    std::cout « "Searching for 30: ";
    if (result) {
        std::cout « "Node was found" « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }

    std::cout « "[\033[0;36m----\033[0m] ";
    intListTest.print();
    result = intListTest.search(10000);
    assertion(result, static_cast<Node<int>*>(nullptr));
    std::cout « "Multiple Item Search -> ";
    std::cout « "Searching for 10000: ";
    if (result) {
        std::cout « "Node was found" « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }
```

```
}


void emptyMerge() {
    LinkedList<int> list1 = LinkedList<int>();
    LinkedList<int> list2 = LinkedList<int>();
    list1.mergeLists(&list2);
    std::string expect_output = "{}";
    std::string output = list1.toString();
    assertion(output, expect_output);
    std::cout « "Empty Merge Output: " « output « std::endl;
}

void callingListEmptyMerge() {
    LinkedList<int> list1 = LinkedList<int>();
    LinkedList<int> list2 = LinkedList<int>();
    list2.add(25);
    list1.mergeLists(&list2);
    std::string expect_output = "{25}";
    std::string output = list1.toString();
    assertion(output, expect_output);
    std::cout « "Calling List Empty Output: " « output « std::endl;
}


void paramterListEmptyMerge() {
    LinkedList<int> list1 = LinkedList<int>();
    LinkedList<int> list2 = LinkedList<int>();
    list1.add(25);
    list1.mergeLists(&list2);
    std::string expect_output = "{25}";
    std::string output = list1.toString();
    assertion(output, expect_output);
    std::cout « "Parameter List Empty Output: " « output « std::endl;
}

void twoNonEmptyMerge() {
    LinkedList<int> list1 = LinkedList<int>();
    LinkedList<int> list2 = LinkedList<int>();
    list1.add(65);
    list1.add(25);
    list1.add(35);
    list2.add(45);
    list2.add(90);
    list2.add(10);
    list1.mergeLists(&list2);
    std::string expect_output = "{10, 25, 35, 45, 65, 90}";
    std::string output = list1.toString();
    assertion(output, expect_output);
    std::cout « "Two Non Empty List Output: " « output « std::endl;
}

void testMerge() {
    emptyMerge();
    callingListEmptyMerge();
    paramterListEmptyMerge();
    twoNonEmptyMerge();
}

void testMergeSort() {
    std::string stringItems[26] = {
        "Quebec", "Victor", "November", "Mike", "Charlie", "X-Ray",
        "Zulu", "Yankee", "Juliett", "Uniform", "Oscar", "Lima", "Romeo",
        "Bravo", "Tango", "Kilo", "Foxtrot", "India", "Delta", "Sierra",
        "Golf", "Alpha", "Papa", "Echo", "Hotel", "Whiskey"
    };
    LinkedList<std::string> stringList = LinkedList<std::string>();

    for (unsigned int i = 0; i < 26; ++i) {
        stringList.add(stringItems[i]);
    }

    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};
    LinkedList<int> intList = LinkedList<int>();

    for (unsigned int i = 0; i < 26; ++i) {
        intList.add(intItems[i]);
    }

    std::cout « "[\033[0;36m----\033[0m] ";
    std::cout « "Before Sort: ";
    stringList.print();

    stringList.mergeSort();

    assertion(stringList.toString(), std::string("{\"Alpha\", \"Bravo\", \"Charlie\", \"Delta\",
\"Echo\", \"Foxtrot\", \"Golf\", "
                                                 "\"Hotel\", \"India\", \"Juliett\", \"Kilo\",
\"Lima\", \"Mike\", \"November\", \"Oscar\", \"Papa\", \"Quebec\", \"Romeo\", "
                                                 "\"Sierra\", \"Tango\", \"Uniform\", \"Victor\",
\"Whiskey\", \"X-Ray\", \"Yankee\", \"Zulu\"}"));
    std::cout « "After Sort: ";
    stringList.print();

    std::cout « std::endl;

    std::cout « "[\033[0;36m----\033[0m] ";
    std::cout « "Before Sort: ";
    intList.print();

    intList.mergeSort();

    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}"));
    std::cout « "After Sort: ";
    intList.print();
}


void testAddFromFile() {
    std::string fileName = "file333.txt";
    LinkedList<std::string>* list = new LinkedList<std::string>;
    Node<std::string>* node = new Node<std::string>;

    list->add("The");
    list->add("best");
    list->add("team");
    list->add("in");
    list->add("CSC340");

    std::cout « "Before adding data from file: " « list->toString() « std::endl;

    list->addFromFile(fileName);

    std::cout « "After adding data from file: " « list->toString() « std::endl;

    node = list->search("Dummy");

    if (node->getNextNode()->getData() == "Text") {
        std::cout « "Test passed" « std::endl;
    }
    else {
        std::cerr « "Test failed" « std::endl;
    }
}

std::string read_file() {
    std::string data = "";
    std::ifstream file;
    std::string line = "";
    file.open("file333.txt");

    if (!file.is_open()) {
        std::cerr « "Error opening file" « std::endl;
    }

    while (file » line) {
        data += line;
    }

    file.close();

    return data;
}

void testBubbleSort() {
    LinkedList<int> intList;
    LinkedList<std::string> stringList;

    int intItems[10] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
    for (unsigned int i = 0; i < 10; ++i) {
        intList.add(intItems[i]);
    }

    std::cout « "[\033[0;36m----\033[0m] ";
    std::cout « "Before Sorting: " « intList.toString() « std::endl;
    intList.bubbleSort();
    assertion(intList.toString(), std::string("{1, 1, 2, 3, 3, 4, 5, 5, 6, 9}"));
    std::cout « "After Sorting: " « intList.toString() « std::endl;

    std::string stringItems[4] = {"zzz", "bbb", "eee", "ddd"};
    for (unsigned int i = 0; i < 4; ++i) {
        stringList.add(stringItems[i]);
    }

    std::cout « "[\033[0;36m----\033[0m] ";
    std::cout « "Before Sorting: " « stringList.toString() « std::endl;
    stringList.bubbleSort();
    assertion(stringList.toString(), std::string("{\"bbb\", \"ddd\", \"eee\", \"zzz\"}"));
    std::cout « "After Sorting: " « stringList.toString() « std::endl;
}

template<class T>
bool assertion(T actual, T expected) {
    if (actual == expected) {
        std::cout « "[\033[0;32mPass\033[0m] ";
        return true;
    }
    else {
        std::cout « "[\033[0;31mFail\033[0m] ";
        std::cout « "Expected: " « expected « std::endl;
        std::cout « "[\033[0;31m--->\033[0m] ";
        std::cout « "  Actual: " « actual « std::endl;
        std::cout « "[\033[0;36m----\033[0m] ";
        return false;
    }
}

template<class T>
bool assertion(T actual, T expected, bool message) {
    if (message) {
        return assertion(actual, expected);
    }
    else {
        return actual == expected;
    }
}

void testInsert() {
    LinkedList<int> intList;
    LinkedList <std::string> stringList;

    // pi
    int intItems[9] = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    for (unsigned int i = 0 ; i < 9; ++i) {
        intList.insert(intItems[i]);
        std::cout « "[\033[0;36m----\033[0m] ";
        intList.print();
    }

    intList.insert(3);
    assertion(intList.toString(), std::string("{1, 1, 2, 3, 3, 4, 5, 5, 6, 9}"));
    intList.print();

    std::string stringItems[2] = {"pug", "bear"};
    for (unsigned int i = 0; i < 2; ++i) {
        stringList.insert(stringItems[i]);
        std::cout « "[\033[0;36m----\033[0m] ";
        stringList.print();
    }

    stringList.insert("zebra");
    assertion(stringList.toString(), std::string("{\"bear\", \"pug\", \"zebra\"}"));
    stringList.print();
}

void testBinarySearch() {
    LinkedList<int> listForSearch = LinkedList<int>();
    Node<int>* result;

    std::cout « "[\033[0;36m----\033[0m] ";
    listForSearch.print();
    result = listForSearch.binarySearch(2);
    assertion(result, static_cast<Node<int>*>(nullptr));
    std::cout « "Empty Binary Search -> ";
    std::cout « "Searching for 2: ";
    if (result) {
        std::cout « "Node was found" « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }

    listForSearch.add(2);
    std::cout « "[\033[0;36m----\033[0m] ";
    listForSearch.print();
    result = listForSearch.binarySearch(2);
    assertion(result->getData(), 2);
    std::cout « "One Item Binary Search -> ";
    std::cout « "Searching for 2: ";
    if (result) {
        std::cout « "Node was found" « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }

    listForSearch.insert(1);
    std::cout « "[\033[0;36m----\033[0m] ";
    listForSearch.print();
    result = listForSearch.binarySearch(1);
    assertion(result->getData(), 1);
    std::cout « "Two Item List Binary Search - First Node -> ";
    std::cout « "Searching for 1: ";
    if (result) {
        std::cout « "Node was found" « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }

    listForSearch.insert(10);
    std::cout « "[\033[0;36m----\033[0m] ";
    listForSearch.print();
    result = listForSearch.binarySearch(10);
    assertion(result->getData(), 10);
    std::cout « "Tree Item List Binary Search - Last Node -> ";
    std::cout « "Searching for 10: ";
    if (result) {
        std::cout « "Node was found with a " « result->getData() « std::endl;
    }
    else {
        std::cout « "Node not found" « std::endl;
    }
}

void testLinkedList() {
    std::cout « " -- Add and Remove Node Test -- " « std::endl;
    testAddRemove();
    std::cout « std::endl;

    std::cout « " -- Insert Node Test -- " « std::endl;
    testInsert();
    std::cout « std::endl;

    std::cout « " -- Search For Node Test -- " « std::endl;
    testSearch();
    std::cout « std::endl;

    std::cout « " -- Binary Search Test -- " « std::endl;
    testBinarySearch();
    std::cout « std::endl;

    std::cout « " -- Merge LinkedLists Test -- " « std::endl;
    testMerge();
    std::cout « std::endl;

    std::cout « " -- Bubble Sort Test -- " « std::endl;
    testBubbleSort();
    std:: cout « std::endl;

    std::cout « " -- Merge Sort Test -- " « std::endl;
    testMergeSort();
    std:: cout « std::endl;
}

void demo() {

    LinkedList<Vault> BankSystem = LinkedList<Vault>();
    LinkedList<Vault> BankSystem2 = LinkedList<Vault>();
    BankSystem2.add(Vault(100));
    BankSystem2.add(Vault(100000));
    BankSystem2.add(Vault(13));
    int choice = 0;
    while (choice != -1) {
        std::cout « "Banking System Main Menu: " « std::endl;
        std::cout « "Enter ’1’ to print the current Bank System" « std::endl;
        std::cout « "Enter ’2’ to add a vault to the current system" « std::endl;
        std::cout « "Enter ’3’ to search for a vault" « std::endl;
        std::cout « "Enter ’4’ to do a binary search must be sorted" « std::endl;
        std::cout « "Enter ’5’ to bubble sort the Bank System" « std::endl;
        std::cout « "Enter ’6’ to merge sort the Bank System" « std::endl;
        std::cout « "Enter ’7’ to merge another Bank System into this one" « std::endl;
        std::cout « "Enter ’-1’ to exit the management system" « std::endl;
        std::cin » choice;
        switch (choice) {
            case 1:
                std::cout « "Current Bank System: ";
                BankSystem.print();
                std::cout « std::endl;
                break;
            case 2:
                int toAdd;
                std::cout « "Enter the value of the new vault to add to the system: ";
                std::cin » toAdd;
                BankSystem.add(Vault(toAdd));
                std::cout « "Added" « std::endl;
                break;
            case 3:
                int searchTarget;
                std::cout « "Enter the value of the target vault ";
                std::cin » searchTarget;
                if (BankSystem.search(searchTarget)) {
                    std::cout « "Vault located" « std::endl;
                }
                else {
                    std::cout « "Vault not located" « std::endl;
                }
                break;
            case 4:
                int binaryTarget;
                std::cout « "Enter the value of the target vault ";
                std::cin » binaryTarget;
                if (BankSystem.binarySearch(binaryTarget)) {
                    std::cout « "Vault located" « std::endl;
                }
                else {
                    std::cout « "Vault not located" « std::endl;
                }
                break;
            case 5:
                std::cout « "Bubble Sorting" « std::endl;
                BankSystem.bubbleSort();
                std::cout « "Sorted!" « std::endl;
                break;
            case 6:
                std::cout « "Merge Sorting" « std::endl;
                BankSystem.mergeSort();
                std::cout « "Sorted!" « std::endl;
                break;
            case 7:
                std::cout « "Merging other Bank System" « std::endl;
                std::cout « "Other system: ";
                BankSystem2.print();
                BankSystem.mergeLists(&BankSystem2);
                std::cout « "New Merge System: ";
                BankSystem.print();
                break;
            default:
                std::cout « "Enter a value between 1 and 6 or -1";
        }


    }
}

#endif /* FunctionTests_hpp */
```

4.2 LinkedList.cpp

```

#ifndef LINKEDLIST_CPP
#define LINKEDLIST_CPP
#include <fstream>
#include "LinkedList.hpp"

template<class T>
LinkedList<T>::LinkedList(){
    this->head = nullptr;
    this->tail = nullptr;
}

template<class T>
LinkedList<T>::~LinkedList() {
    clear();
}

template<class T>
void LinkedList<T>::clear() {
    Node<T>* nodeToDelete = head;
    while (head != nullptr) {
        head = head->getNextNode();
        delete nodeToDelete;
        nodeToDelete = head;
    }
}

template<class T>
void LinkedList<T>::insert(T data) {
    Node<T>* newNode = new Node<T>(data);
    if (this->head == nullptr) {
        this->head = newNode;
        this->tail = newNode;
        return;
    }
    if (data <= this->head->getData()) {
        newNode->setNextNode(head);
        this->head->setPrevNode(newNode);
        this->head = newNode;
        return;
    }
    if (data >= tail->getData()) {
        newNode->setPrevNode(tail);
        this->tail->setNextNode(newNode);
        this->tail = newNode;
        return;
    }
    Node<T>* temp = this->head;
    while (temp->getNextNode() != nullptr && temp->getNextNode()->getData() < data) {
        temp = temp->getNextNode();

    }

    newNode->setNextNode(temp->getNextNode());
    temp->setNextNode(newNode);
    newNode->setPrevNode(temp);
    newNode->getNextNode()->setPrevNode(newNode);
}

template<class T>
void LinkedList<T>::add(T data) {
    Node<T>* newNode = new Node<T>(data);

    if (this->head == nullptr) {
        this->head = newNode;
        this->tail = newNode;
    }
    else {
        this->tail->setNextNode(newNode);
        newNode->setPrevNode(tail);
        this->tail = newNode;
    }
}

template<class T>
void LinkedList<T>::remove(T data) {
    Node<T>* nodeToDelete = search(data);
    if (nodeToDelete != nullptr) {
        if (nodeToDelete == this->head) {
            this->head = head->getNextNode();
            this->head->setPrevNodeNull();
        }
        else if (nodeToDelete == this->tail) {
            this->tail = tail->getPrevNode();
            this->tail->setNextNodeNull();
        }
        else {
            Node<T>* prevNode = nodeToDelete->getPrevNode();
            Node<T>* nextNode = nodeToDelete->getNextNode();
            prevNode->setNextNode(nextNode);
            nextNode->setPrevNode(prevNode);
        }
        delete nodeToDelete;
    }
}

template<class T>
Node<T>* LinkedList<T>::search(T data) {
    Node<T>* temp = this->head;

    while (temp) {
        if (temp->getData() == data) {
            return temp;
        }

        temp = temp->getNextNode();
    }
    return nullptr;
}

template<class T>
std::string LinkedList<T>::toString() {
    Node<T>* temp = this->head;
    std::string output = "";
    std::string quote = (typeid(T).name() == typeid(std::string("")).name()) ? "\"" : "";

    while (temp != nullptr) {
        T val = temp->getData();
        output += quote + to_string(val) + quote;
        temp = temp->getNextNode();
        if (temp != nullptr) {
            output += ", ";
        }
    }
    return "{" + output + "}";
}

template<class T>
void LinkedList<T>::mergeSort() {
    // base case: 1 or 0 Nodes
    if (this->head == nullptr || this->head->getNextNode() == nullptr) {
        return;
    }

    // split the LinkedList in half
    Node<T>* subHead1 = this->head;
    Node<T>* subHead2 = findMid(this->head, this->tail);
    Node<T>* subTail1 = subHead2->getPrevNode();
    Node<T>* subTail2 = this->tail;

    // detach the two halves
    subTail1->setNextNodeNull();
    subHead2->setPrevNodeNull();

    // recurse first half
    this->head = subHead1;
    this->tail = subTail1;
    mergeSort();
    subHead1 = this->head;
    subTail1 = this->tail;

    // recurse second half
    this->head = subHead2;
    this->tail = subTail2;
    mergeSort();
    subHead2 = this->head;
    subTail2 = this->tail;

    // merge both halves
    this->head = nullptr;
    Node<T>* nodeptr = nullptr;

    // compare head of both halves
    while (subHead1 != nullptr && subHead2 != nullptr) {
        if (subHead1->getData() < subHead2->getData()) {
            if (this->head == nullptr) {
                this->head = subHead1;
                nodeptr = subHead1;
            }
            else {
                nodeptr->setNextNode(subHead1);
                subHead1->setPrevNode(nodeptr);
                nodeptr = subHead1;
            }
            subHead1 = subHead1->getNextNode();
        }
        else {
            if (this->head == nullptr) {
                this->head = subHead2;
                nodeptr = subHead2;
            }
            else {
                nodeptr->setNextNode(subHead2);
                subHead1->setPrevNode(nodeptr);
                nodeptr = subHead2;
            }
            subHead2 = subHead2->getNextNode();
        }
    }

    // add the rest of first half to the main LinkedList
    while (subHead1 != nullptr) {
        if (this->head == nullptr) {
            this->head = subHead1;
            nodeptr = subHead1;
        }
        else {
            nodeptr->setNextNode(subHead1);
            subHead1->setPrevNode(nodeptr);
            nodeptr = subHead1;
        }
        subHead1 = subHead1->getNextNode();
    }

    // add the rest of second half to the main LinkedList
    while (subHead2 != nullptr) {
        if (this->head == nullptr) {
            this->head = subHead2;
            nodeptr = subHead2;
        }
        else {
            nodeptr->setNextNode(subHead2);
            subHead2->setPrevNode(nodeptr);
            nodeptr = subHead2;
        }
        subHead2 = subHead2->getNextNode();
    }

    this->tail = nodeptr;
}

template<class T>
void LinkedList<T>::bubbleSort() {
    // do not sort if empty or one
    if (this->head == nullptr || this->head->getNextNode() == nullptr)
        return;

    bool swap;
    Node<T>* current = this->head;
    Node<T>* sorttail = nullptr;

    while (current != sorttail){
        swap = false;
        Node <T>* current2 = this->head;

        while (current2->getNextNode () != sorttail){
            if (current2->getData() > current2->getNextNode()->getData()){
                T temp = current2 -> getData();
                current2->setData(current2->getNextNode()->getData());
                current2->getNextNode()->setData(temp);
                swap = true;
            }
            current2 = current2->getNextNode();
        }
        sorttail = current2; // update tail to last swap
        if (!swap)
            break; // if no swap the list is already sorted
    }
}

template<class T>
void LinkedList<T>::addFromFile (std::string fileName) {
    std::ifstream file;
    T data;

    file.open(fileName);

    if (file.peek() == std::ifstream::traits_type::eof()) {
        std::cerr « "Error: File is empty" « std::endl;
        exit(1);
    }

    if (!file.is_open()) {
        std::cerr « "Error opening file" « std::endl;
        exit(1);
    }

    while (file » data) {
        if (file.eof()) {
            break;
        }

        this->add(data);
    }

    if (!file.eof()) {
        std::cerr « "Error reaching end of file" « std::endl;
        exit(1);
    }

    file.close();
}

template<class T>
void LinkedList<T>::mergeLists(const LinkedList<T>* listTwo) {
    if (!this->head && !listTwo->head) {
        return;
    }
    else {
        mergeSort();
        Node<T>* temp = listTwo->head;
        while (temp != nullptr) {
            this->insert(temp->getData());
            temp = temp->getNextNode();
        }
    }
}

template<class T>
void LinkedList<T>::print() {
    Node<T>* temp = this->head;

    if (this->head == nullptr) {
        std::cout « "The linked list is empty." « std::endl;
        return;
    }

    while (temp != nullptr) {
        std:: cout « temp->getData() « " ";
        temp = temp->getNextNode();
    }

    std::cout « std::endl;
}

template<class T>
Node<T>* LinkedList<T>::binarySearch(T target) {
    Node<T>* searchHead = this->head;
    Node<T>* searchTail = this->tail;
    Node<T>* searchMid = findMid(searchHead, searchTail);
    if (searchHead) {
        while (searchHead->getData() <= searchTail->getData()) {
            if (target == searchHead->getData()) {
                return searchHead;
            }
            else if (target == searchMid->getData()) {
                return searchMid;
            }
            else if (target == searchTail->getData()) {
                return searchTail;
            }

            if (target < searchMid->getData()) {
                searchHead = searchHead->getNextNode();
                searchTail = searchMid->getPrevNode();
                searchMid = findMid(searchHead, searchTail);
            }
            else if (target < searchMid->getData()) {
                searchHead = searchMid->getNextNode();
                searchTail = searchTail->getPrevNode();
                searchMid = findMid(searchHead, searchTail);
            }
        }
    }

    return nullptr;
}

#endif
```

4.3 LinkedList.hpp

```
//
//  LinkedList.hpp
//  CSC340GP
//
//  Created by e on 5/13/23.
//

#ifndef LINKEDLIST_HPP
#define LINKEDLIST_HPP
#include "Node.hpp"
#include <iostream>
```

```
#include <string>
#include <sstream>

template<class T>
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    void clear();
    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();
    Node<T>* binarySearch(T target);

private:
    Node<T>* head;
    Node<T>* tail;

    std::string to_string(const T& obj) {
        std::ostringstream oss{};
        oss « obj;
        return oss.str();
    }

    static Node<T>* findMid(Node<T>* start, Node<T>* end) {
        bool flip = true;
        while (start != end) {
            if (flip) {
                start = start->getNextNode();
            }
            else {
                end = end->getPrevNode();
            }
            flip = flip ? false : true;
        }

        return start;
    }
};
#include "LinkedList.cpp"
#endif /* LinkedList_hpp */
```

4.4 main.cpp

```
/********************************************/
#include "LinkedList.hpp"
#include "FunctionTests.hpp"

int main(int argc, const char* argv[]) {

    testLinkedList();
    demo();

    return 0;
}
```

4.5 Node.cpp

```

#ifndef NODE_CPP
#define NODE_CPP

#include "Node.hpp"

template<class T>
Node<T>::Node() {
    this->data = "";
    this->next = nullptr;
    this->prev = nullptr;
}

template<class T>
```

```
Node<T>::Node(T data) {
    this->data = data;
    this->next = nullptr;
    this->prev = nullptr;
}

template<class T>
Node<T>::~Node() {
    this->next = nullptr;
    this->prev = nullptr;
}

// getters
template<class T>
T Node<T>:: getData() {return this->data;}

template<class T>
Node<T>* Node<T>::getNextNode() {return this->next;}

template<class T>
Node<T>* Node<T>::getPrevNode() {return this->prev;}

template<class T>
void Node<T>::setData(T data) {this->data = data;}

template<class T>
void Node<T>::setNextNode(Node* next) {this->next = next;}

template<class T>
void Node<T>::setPrevNode(Node* prev) {this->prev = prev;}

template<class T>
void Node<T>::setNextNodeNull() {this->next = nullptr;}

template<class T>
void Node<T>::setPrevNodeNull() {this->prev = nullptr;}
#endif
```

4.6 Node.hpp

```
//
//  Node.hpp
//  CSC340GP
//
//  Created by e on 5/13/23.
//

#ifndef NODE_HPP
#define NODE_HPP

template<class T>
class Node {
public:
    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();

private:
    T data;
    Node<T>* next;
    Node<T>* prev;
};
#include "Node.cpp"
#endif /* Node_hpp */
```

4.7 Vault.cpp

```
//
//  Vault.cpp
//  CSC340GP
//
//  Created by e on 5/18/23.
//
#ifndef Vault_cpp
#define Vault_cpp

#include "Vault.hpp"

Vault::Vault() {
    this->balance = 0;
}

Vault::Vault(int startBal) {
    this->balance = startBal;
}

Vault::~Vault() {}

bool Vault::operator==(const Vault& r) {
    return this->balance == r.balance;
}

bool Vault::operator!=(const Vault& r) {
    return this->balance != r.balance;
}

bool Vault::operator<(const Vault& r) {
    return this->balance < r.balance;
}

bool Vault::operator>(const Vault& r) {
    return this->balance > r.balance;
}

bool Vault::operator<=(const Vault& r) {
    return this->balance <= r.balance;
}

bool Vault::operator>=(const Vault& r) {
    return this->balance >= r.balance;
};
#endif
```

4.8 Vault.hpp

```
//
//  Vault.hpp
//  CSC340GP
//
//  Created by e on 5/18/23.
//

#ifndef Vault_hpp
#define Vault_hpp

class Vault {
public:
    Vault();
    Vault(int startBal);
    ~Vault();

    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);

private:
    int balance;
    friend std::ostream& operator«(std::ostream& os, const Vault& v) {
        os « "Vault with balance: " « v.balance;
        return os;
    }
};
#include "Vault.cpp"
#endif /* Vault_hpp */
```

Index