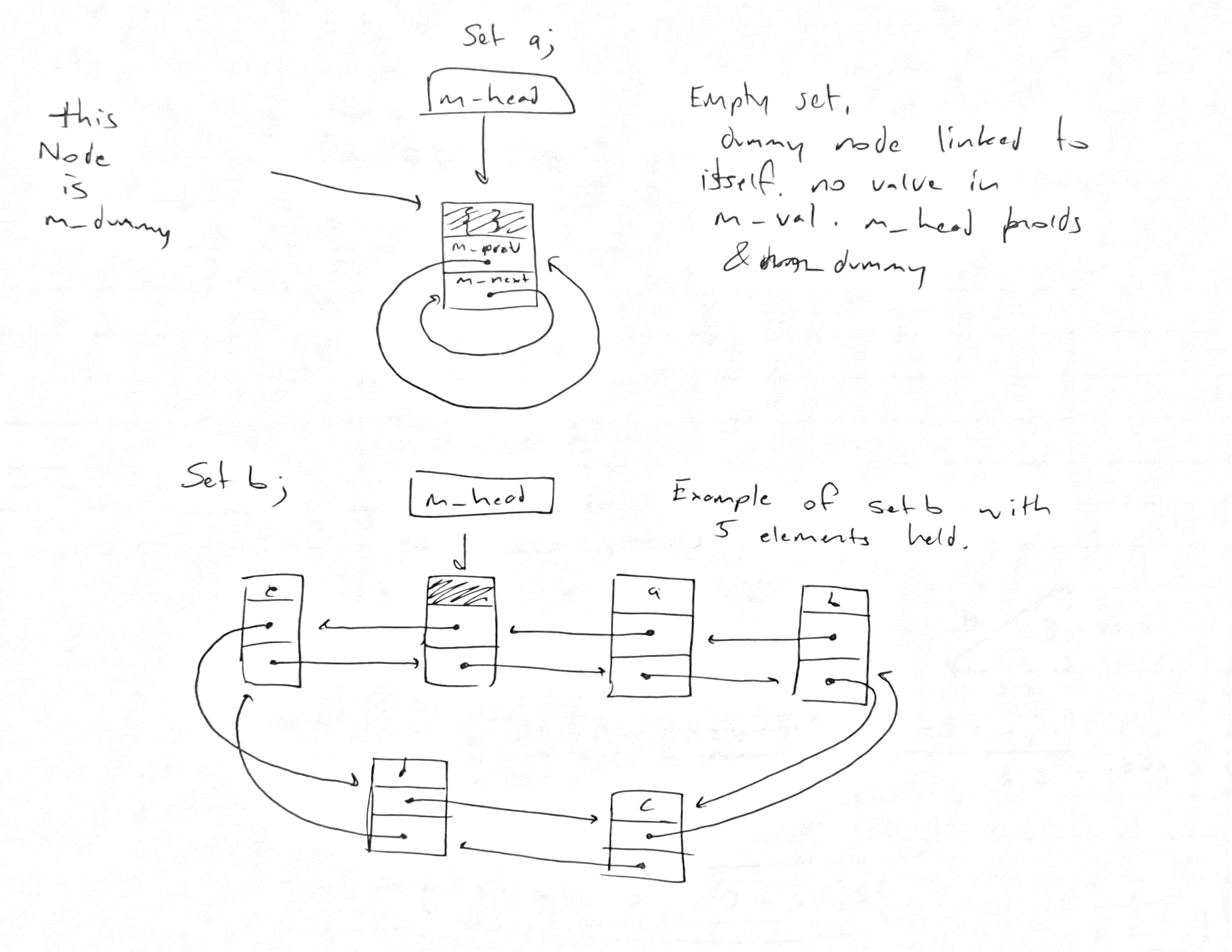
Ryan Foundoulis

605107291

Project 2 Report

1. Doubly Linked List Description

I set my doubly linked list up so that it will be circularly linked with the “start” (or end) of our listed being a dummy node which holds no value, but it will be (or is) linked to the first and last nodes to create a circular list. We also have a head pointer which will always hold the address of the dummy node, allowing for easy access to the start or end of the list. Each node holds a value of ItemType (whatever is specified in Set.h) and pointers to the nodes before and after it in the list. If it is the first or last (non-dummy) node, one of its pointers will be connected to the dummy (creating the double circular link). The nodes are inserted at the end of the list regardless of the value (so in chronological order) but this does not matter as the get is set up to output value from smallest to largest as you count upwards (i.e. 0,1,2,3,4…).



2. Pseudocode:

Set::~Set() {

If the set is not empty {

Create a pointer to a node called “current” addressed to the first (non-dummy node)

Until iterate current through the list and current points to dummy node {

Create a pointer to node called “temp” at the same address as current;

Move current to the next node in the list;

Delete node that temp points to

}

}

Delete the dummy node

}

bool Set::insert(const ItemType& value) {

Create a pointer called “temp” to the first non-dummy node;

While temp does not point to dummy {

Create ItemType “x” set to the value held by the node temp points to;

If “x” equals “value” {

Return false (break from function);

}

Set temp to the address of the next node in the list;

}

Create newNode that holds “value”, with prev pointer set to dummy’s prev pointer and its next pointer set to &dummy;

If the set is empty

Set dummy’s next pointer to newNode;

Else

Set node immediately before newNode’s next pointer to newNode:

Set dummy’s prev pointer to newNode;

Increase the set’s size count by one;

Return true;

}

bool Set::erase(const ItemType& value) {

Create Node\* p set to first node and iterate through the whole list and for each node {

If the value store in node where p points equals “value” {

Create Node\* q set to the node before where p is;

Set next pointer of node q to the next pointer of node p;

Set the prev pointer of the node after p to q;

Delete p;

Decrease size of Set;

Return true;

}

}

Return false;

bool Set::get(int pos, ItemType& value) const {

If “pos” is less than zero or greater than or equal to size of set

Return false;

Create Node\* p set to first node and iterate until it reaches the dummy node {

Create count set to 0;

Create Node\* q set to first node and iterate until it reaches the dummy node {

If value held by p is greater than the value held by q

Increase count by one;

}

If count is equal to pos {

Set “value” to value held by the node p points to;

Return true;

}

}

Return false;

}

void Set::swap(Set& other) {

Create Node\* temp pointing to this head pointer;

Set head pointer of this set to head pointer of other set;

Set head pointer of other set to temp pointer;

Create int b set to this sets size;

Set this sets side to other sets size;

Set other sets size to b;

}

void unite(const Set& s1, const Set& s2, Set& result) {

Create set tempSet;

While iterating int i from 0 to 1 less than size of set {

Store value of i’th node (from get()’s ordering) into “x”

Insert “x” value into tempSet;

}

While iterating int i from 0 to 1 less than size of set {

Store value of i’th node (from get()’s ordering) into “y”

Insert “y” value into tempSet if it’s not already present;

}

Swap “result” with tempSet;

}

void subtract(const Set& s1, const Set& s2, Set& result) {

Copy construct Set temp from s1

While iterating int i from 0 to 1 less than size of set {

Store value of i’th node (from temp.get()’s ordering) into “x”

While iterating int i from 0 to 1 less than size of set {

Store value of i’th node (from S2.get()’s ordering) into “y”

If “x” is equal to “y” {

Erase the node that holds x from temp

}

}

}

Set result set to temp set;

}

3. Test Codes:

#include "Set.h"

#include <iostream>

#include <cassert>

using namespace std;

void check1() { // Checks functions within single Set

// Mainly focusing on size,empty,contains,get, and erase functions

Set ll; // checks to see that all these implementations do not cause undefined behaviors

assert(ll.empty()); // Tests empty function correctness

assert(ll.size() == 0); // Tests size on empty list

assert(ll.insert("a"));

assert(ll.contains("a")); // Test contain

assert(ll.size() == 1);

assert(!ll.empty()); // Tests empty on non-empty function

assert(ll.insert("b"));

assert(ll.contains("b"));

assert(ll.size() == 2);

ItemType a;

ItemType b;

assert(ll.get(1, b) && b == "b"); // Tests get function ordering

assert(ll.get(0, a) && a == "a");

assert(ll.erase("a")); // Makes sure erase works

assert(!ll.contains("a")); // Make sure contains returns false when it should

assert(ll.get(0,b) && b == "b"); // Tests for correct get ordering when a node is deleted

assert(ll.erase("b"));

assert(!ll.erase("b")); // Tests that erase correctly does nothing

assert(!ll.contains("b"));

assert(ll.insert("a")); // Fill list with 5 values

assert(ll.insert("b"));

assert(ll.insert("c"));

assert(ll.insert("d"));

assert(ll.insert("e"));

assert(!ll.insert("a")); // Will not double insert

assert(!ll.insert("b")); // Will not double insert

assert(ll.size() == 5);

assert(ll.get(0, b) && b == "a"); // Test get ordering again

assert(ll.get(1, b) && b == "b");

assert(ll.get(2, b) && b == "c");

assert(ll.get(3, b) && b == "d");

assert(ll.get(4, b) && b == "e");

while(ll.size() > 0) { // Checks if Iterating through all i's does not cause issues

int i = ll.size() - 1; // With node linking. That deleting nodes out of order will not cause issues

ll.get(i, b);

ll.erase(b);}

assert(ll.size() == 0); // Test all nodes correctly deleted

assert(ll.empty()); // Tests that deallocating an empty set does not cause problems

}

void check2() { // Checks copy constructor, assignment operator, swap function

Set ii; Set ll; Set jj; // Creating empty sets

jj,swap(ll); // Test empty swaps

assert(ll.insert("a")); // Fill ll

assert(ll.insert("b"));

assert(ll.insert("c"));

assert(ll.insert("d"));

assert(ll.insert("e"));

jj.swap(ll); // Test swap with one empty list

assert(jj.size() == 5); // Tests for correct size after swapping

assert(ll.size() == 0);

assert(!jj.insert("a")); // Test that front of list is properly search

assert(!jj.insert("e")); // Tests back of list to make sure it is properly searched

ii = jj;

assert(ii.size() == 5); // Tests that assignment operator transfers size

assert(ll.insert("a")); // Fill ll list

assert(ll.insert("b"));

assert(ll.insert("c"));

assert(ll.insert("d"));

assert(ll.insert("e"));

assert(ll.erase("a"));

assert(ll.erase("b")); // Erasing through all of the list to ensure list relinking

assert(ll.erase("c")); // and counting are not happening wrong

assert(ll.erase("d"));

assert(ll.erase("e"));

assert(ll.empty());

assert(!ii.empty());

assert(!jj.empty());

ii = jj = ll; // Test that assignment operator works with multiple

assert(ll.empty() // Assignments in a single line

&& ii.empty()

&& jj.empty());

assert(ll.insert("a"));

assert(ll.insert("b"));

assert(ll.insert("c"));

assert(ll.insert("d"));

assert(ll.insert("e"));

jj.swap(ll);

ii = ll = jj; // Checks that ii,jj,ll are all destructed when containing nodes

}

void check3() {

Set ii; Set ll; Set jj;

assert(ll.insert("a")); // These next 12 lines are to set up for unite and subtract testing

assert(ll.insert("b"));

assert(ll.insert("c"));

assert(ll.insert("d"));

assert(ll.insert("e"));

assert(ll.size() == 5);

assert(jj.insert("a"));

assert(jj.insert("b"));

assert(jj.insert("c"));

assert(jj.size() == 3);

assert(ii.insert("b"));

assert(ii.insert("c"));

assert(ii.size() == 2);

unite(ii, jj, ii);

assert(ii.size() == 3);

subtract(ll, ii, ll);

assert(ll.size() == 2);

unite(ll, jj, ll);

assert(ll.size() == 5);

subtract(ii, ll, ii); // Checks that count won't go negative and that the program will

subtract(jj, ll, jj); // Not attempt to erase what is not there

assert(ii.empty()

&& jj.empty());

unite(ii, jj, ii); // Test on empty sets

assert(ii.empty());

subtract(ii, jj, ii); // Test on empty sets

subtract(ii, ll, ii);

assert(ii.size() == 0);

subtract(ll, ii, ll); // Subtract empty from non-empty

subtract(ll, ii, jj); // Tests returning to different set

assert(ll.size() == 5

&& jj.size() == 5);

}

int main()

{

check1();

check2();

check3();

cerr << "All test passed" << endl;

}