CS32: Programming Assignment 2

Seeing Double

Yaman Yucel: 605704529

Description of the design of your doubly-linked list implementation: Each node consists of pointer to previous and next node and also holds a specific value. Each set consists of a pointer to head of the list called as m\_head. To reach size quickly, m\_size parameter is used. M\_size parameter holds the number of items in the set. Set items are arranged in increasing order to satisfy get function requirement. The list is **not** circular and does **not** contain dummy node. List is regular doubly linked list.

Empty Set:

M\_head is nullpointer.



One Item Set:

M\_Head points to a node. m\_head -> m\_next is nullptr, m\_head -> m\_prev is nullptr.



N element Set:

m\_head -> m\_prev is nullptr. Last Node’s->m\_next is nullptr.



Pseudocode for algorithms:

**~Set()**

Assign head to a temporary pointer.

Iterate through list until reaching end

Save next object’s pointer

Delete current object

Restore next object’s pointer to current object.

**Set::Set(const Set& other)**

Initialize set as empty set

If other is not empty set

Create a temporary node pointer and assign other’s head to iterate other set

Copy first node

Iterate to other set’s next node

If other set’s next node does not exist, return after setting m\_head -> nullptr.

Else,

Create a temporary node pointer(current) and assign this’s head to iterate this set.

Iterate to other set’s nodes until there is no node

Create new node to current node’s next node

Iterate current node to next node and set it’s value

Iterate to other set’s next node

Set last node’s next to nullptr.

Set size

**Set& Set::operator=(const Set& rhs)**

Check whether rhs == lhs, if they are same return current set.

Else,

Create a temporary set using copy constructor on rhs

Swap pointer and m\_size with this and temporary set

Return value inside this.

**bool Set::empty() const**

Check m\_size == 0

**Int Set::size() const**

Return m\_size

**Bool set::insert(const ItemType& value)**

Check whether value exists with contains(value), if yes return false

Else,

Check whether list is empty, if yes add first node by simply assigning m\_head a pointer.

Increase size and return true

If list is not empty check, whether we will insert to head

If yes, insert a new node to beginning

Create a new node

Set it’s value

Assign it’s next pointer to m\_head

Assign next Node’s prev to the node that is being inserted.

Assign head pointer to the node that is being inserted.

Assign prev of new node to be a nullptr

Increase size and return true

If we will insert at middle or end, continue from here

Create an iterator to find the node which you will insert after that node.

Find that node, if you can not find the node, set current to the node at the end

Create the new node to be inserted

Set it’s value

Set new node’s next to be current’s next, if we are at the end nullptr will be assigned.

If we are not inserting at the end, set the node being inserted’s next node’s prev the node being inserted.

Set current nodes next, the node being inserted.

Set the node being inserted’s prev the current node.

Increase size, return true

**Bool Set::erase(const ItemType& value)**

Start by finding the node to be deleted.

Set temporary node to head

Search for the node to be deleted.

If node is not found

Return false

If we are going to delete the first node

Detach node to be deleted from head pointer and set it to next node or nullptr if there is only one element left.

If we are not deleting the node at the end,

Set next node after the being deleted node’s prev to be the being deleted node’s prev

If we are not deleting the node at the beginning of the list,

Set prev node after the being deleted node’s next to be the being deleted node’s next

Delete the selected node

Decrease size and return true

**bool Set::contains(const ItemType& value) const**

Iterate through all list

Return true if the given value is found

If not found or empty list return false,

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

Check given pos is in the bounds,

Iterate through list until you reach the pos – 1 the element.

Set value with the node’s value and return true

If pos is not in the bounds return false.

Void Set::swap(Set& other)

if this and other is not same,

swap m\_head pointer

swap m\_size

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

Check whether s1,s2 and result are different sets with different memory locations,

if they are all different sets

empty result set

add s1 elements to result set

add s2 elements to result set

If above false, check whether s1 and result are same s2 different

Add s2 elements to result set

If above false, check whether s2 and result are same s1 different

Add s1 elements to result set

If above false, check whether s1 and s2 are same result different

Empty result set

Add s1 elements to result set

If above false, check whether s1 and s2 and result are same.

Return do not do anything.

**void butNot(const Set&s1, const Set&s2, Set& result)**

Check whether s1,s2 and result are different sets with different memory locations,

if they are all different sets

empty result set

add s1 elements to result set

remove s2 elements from result set

If above false, check whether s1 and result are same s2 different

discard s2 elements from result set

If above false, check whether s2 and result are same s1 different

Copy results to another temporary set

Empty result set

Add s1 elements to result set

Remove copied set’s elements from result set

If above false, check whether s1 and s2 are same result different

Clear result set

Return empty set since s1 butNot s1 is empty set

If above false, check whether s1 and s2 and result are same.

Clear result set

Return empty set since s1 butNot s1 is empty set

**List of test cases: I have used ItemType = double to set my set implementation**

#include "Set.h"

#include <iostream>

#include <string>

#include <cassert>

int main()

{

Set test;

//try to delete an element from empty list

assert(!test.erase(5));

ItemType tmp;

// insert element to an empty list

test.insert(5);

tmp = 123123;

assert(test.get(0, tmp) && tmp == 5);

// insert element to an empty list

// insert element to start one-element list

test.insert(4);

tmp = 123123;

assert(test.get(0, tmp) && tmp == 4);

// insert element to end 2-element list

test.insert(6);

tmp = 123123;

assert(test.get(2, tmp) && tmp == 6);

test.insert(5.5);

tmp = 123123;

assert(test.get(2, tmp) && tmp == 5.5); // insert to middle

test.erase(4); // erase the head

tmp = 123123;

assert(test.get(0, tmp) && tmp != 4);

//erase the end

test.erase(6);

tmp = 123123;

assert(!test.get(2, tmp));

//size check

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

test.erase(5);// delete remaining

test.erase(5.5);

assert(test.empty()); // check empty

Set empty(test); // create empty set using test

assert(empty.empty());

test.insert(1);

test.insert(2);

Set nonempty(test); // testing copy constructor for nonempty set and also size()

tmp = 123123;

assert(nonempty.get(0, tmp) && tmp == 1 && nonempty.get(1,tmp) && tmp == 2 && nonempty.size() == 2);

test.erase(1);

Set oneelement(test); // use copy constructor for one element set

tmp = 123123;

assert(oneelement.get(0, tmp) && tmp == 2 && oneelement.size() == 1);

//contains test, empty set, 1 element set, 2 element set

assert(!empty.contains(1));

assert(oneelement.contains(2));

assert(!oneelement.contains(10));

assert(nonempty.contains(1));

assert(!nonempty.contains(10));

Set s; // insert 5 elements and check with dump()

s.insert(5);

s.insert(4);

s.insert(3);

s.insert(2);

s.insert(1);

s.dump(); // s = 1 2 3 4 5

s.erase(5);

s.erase(4);

s.erase(3);

s.erase(2);

s.erase(1);

s.dump(); // s = // erase 5 elements and check with dump()

Set sa;

sa.insert(5);

sa.insert(4);

sa.insert(3);

sa.insert(2);

sa.insert(1); // sa = 1 2 3 4 5

Set sb = sa; // check assignment operator

Set sc;

sc = sa; // check assignment operator

sc.dump();

sb.dump();

ItemType d = 10;

sa.get(0, d); //check get function

std::cerr <<"Retrieved value is " << d << std::endl;

Set a;// check the given example for unite

a.insert(2);

a.insert(8);

a.insert(3);

a.insert(9);

a.insert(5);

a.dump();

Set b;

b.insert(6);

b.insert(3);

b.insert(8);

b.insert(5);

b.insert(10);

b.dump();

Set r;

r.insert(100);

r.insert(101);

r.dump();

unite(a, b, r);

r.dump();// check the given example for unite

butNot(a, b, r);

r.dump();// check the given example for butNot

//unite and butNot functions test

Set reload\_s1;

reload\_s1.insert(1);

reload\_s1.insert(2);

Set reload\_s2;

reload\_s2.insert(2);

reload\_s2.insert(3);

reload\_s2.insert(4);

Set reload\_result;

reload\_result.insert(5);

reload\_result.insert(6);

Set s1;

Set s2;

Set result;

ItemType temp = 32177321;

//branches of unite are tested for all empty set and all non-empty set

unite(s1, s2, result); // s1 != result && s1 != s2 && s2 != result

assert(result.empty());

unite(s1, s2, s1);// s1 == result && s1 != s2 && s2 != result

assert(s1.empty());

unite(s1, s2, s2);// s1 != result && s1 != s2 && s2 == result

assert(s2.empty());

unite(s1, s1, result); // s1 != result && s1 == s2 && s2 != result

assert(result.empty());

unite(s1, s1, s1);// s1 == result && s1 == s2 && s2 == result

assert(s1.empty());

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

unite(s1, s2, result); // s1 != result && s1 != s2 && s2 != result

assert(result.get(0, temp) && temp == 1 && result.size() == 4);

temp = 32177321;

assert(result.get(1, temp) && temp == 2 && result.size() == 4);

temp = 32177321;

assert(result.get(2, temp) && temp == 3 && result.size() == 4);

temp = 32177321;

assert(result.get(3, temp) && temp == 4 && result.size() == 4);

temp = 32177321;

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

unite(s1, s2, s1);// s1 == result && s1 != s2 && s2 != result

assert(s1.get(0, temp) && temp == 1 && s1.size() == 4);

temp = 32177321;

assert(s1.get(1, temp) && temp == 2 && s1.size() == 4);

temp = 32177321;

assert(s1.get(2, temp) && temp == 3 && s1.size() == 4);

temp = 32177321;

assert(s1.get(3, temp) && temp == 4 && s1.size() == 4);

temp = 32177321;

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

unite(s1, s2, s2);// s1 != result && s1 != s2 && s2 == result

assert(s2.get(0, temp) && temp == 1 && s2.size() == 4);

temp = 32177321;

assert(s2.get(1, temp) && temp == 2 && s2.size() == 4);

temp = 32177321;

assert(s2.get(2, temp) && temp == 3 && s2.size() == 4);

temp = 32177321;

assert(s2.get(3, temp) && temp == 4 && s2.size() == 4);

temp = 32177321;

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

unite(s1, s1, result); // s1 != result && s1 == s2 && s2 != result

assert(result.get(0, temp) && temp == 1 && result.size() == 2);

temp = 32177321;

assert(result.get(1, temp) && temp == 2 && result.size() == 2);

temp = 32177321;

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

unite(s1, s1, s1);// s1 == result && s1 == s2 && s2 == result

assert(s1.get(0, temp) && temp == 1 && s1.size() == 2);

temp = 32177321;

assert(s1.get(1, temp) && temp == 2 && s1.size() == 2);

temp = 32177321;

Set emt;

s1 = emt;

s2 = emt;

result = emt;

assert(s1.empty());

assert(s2.empty());

assert(result.empty());

butNot(s1, s2, result); // s1 != result && s1 != s2 && s2 != result

assert(result.empty());

butNot(s1, s2, s1);// s1 == result && s1 != s2 && s2 != result

assert(s1.empty());

butNot(s1, s2, s2);// s1 != result && s1 != s2 && s2 == result

assert(s2.empty());

butNot(s1, s1, result); // s1 != result && s1 == s2 && s2 != result

assert(result.empty());

butNot(s1, s1, s1);// s1 == result && s1 == s2 && s2 == result

assert(s1.empty());

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

butNot(s1, s2, result); // s1 != result && s1 != s2 && s2 != result

assert(result.get(0, temp) && temp == 1 && result.size() == 1);

temp = 32177321;

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

butNot(s1, s2, s1);// s1 == result && s1 != s2 && s2 != result

assert(s1.get(0, temp) && temp == 1 && s1.size() == 1);

temp = 32177321;

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

butNot(s1, s2, s2);// s1 != result && s1 != s2 && s2 == result

assert(s2.get(0, temp) && temp == 1 && s2.size() == 1);

temp = 32177321;

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

butNot(s1, s1, result); // s1 != result && s1 == s2 && s2 != result

assert(result.empty());

s1 = reload\_s1;

s2 = reload\_s2;

result = reload\_result;

butNot(s1, s1, s1);// s1 == result && s1 == s2 && s2 == result

assert(s1.empty());

std::cerr << "All tests passed" << std::endl;

}