Tanzeela Khan

204577214

Project 2: Report

1. Short description of my implementation of a doubly linked list:

My doubly-linked list is linear/non-circular. The head’s prev pointer is null while its next pointer points to the next node. The last node’s next pointer points to null while it’s previous pointer points to the node right before it. In an empty list, the head’s nextPtr is null. Each node structure has a key, value, nextpr, and prevptr. Items are added to the front by reassigning the headptr. I do not use a tailptr.

1. Pseudocode for non-trivial algorithms:

    Map()         // Create an empty map (i.e., one with no key/value pairs)

     set current amount and headPtr to nothingness

    Map(const Map& old)

create ptr curr that holds old’s headptr

if old list is empty

return

run through curruntil at last node

run though curr backwards using prev

insert key/value of old’s into our's

change size of currentAmt that old's

    //When a brand new Map is created as a copy of an existing Map, enough new nodes must be allocated to hold a duplicate of the original list.

    Map& operator=(const Map& old)

run through our map

use destruction operation to erase nodes

change size of currentAmt that old's

use copy constructor for old’s “daNew”

set daNew's size to zero and set its headptr to null

return \*this

    //When an existing Map (the left-hand side) is assigned the value of another Map (the right-hand side), the result must be that the left-hand side object is a duplicate of the right-hand side object, with no memory leak of list nodes (i.e. no list node from the old value of the left-hand side should be still allocated yet inaccessible).

    bool empty()const  // Return true if the map is empty, otherwise false.

     if current amount is 0

return true

return false

    int size()const    // Return the number of key/value pairs in the map.

return current amount

    bool insert(const KeyType& key, const ValueType& value)

assign headPtr to “daSearch”

if headPtr is null

create new node p key/value

p’s next and prev are set to null

headPtr is set to p and current amount is incremented

returns true;

run through list

if any key in the list equals the key we are inserting, return false

create a new node p key/value

p prepares to become “head” by setting its next to head and its prey to null

head’s pre is set to p and head is finally set to p;

increment currentAmount and return true

    // If key is not equal to any key currently in the map, and if the

    // key/value pair can be added to the map, then do so and return true.

    // Otherwise, make no change to the map and return false (indicating

    // that either the key is already in the map, or the map has a fixed

    // capacity and is full).

    bool update(const KeyType& key, const ValueType& value)

use pointer to equal search function to find key

if not found, return

otherwise, will set pointer’s value to designated value

    // If key is equal to a key currently in the map, then make that key no

    // longer map to the value it currently maps to, but instead map to

    // the value of the second parameter; return true in this case.

    // Otherwise, make no change to the map and return false.

    bool insertOrUpdate(const KeyType& key, const ValueType& value)

try update or try insert and return true

return false

    // If key is equal to a key currently in the map, then make that key no

    // longer map to the value it currently maps to, but instead map to

    // the value of the second parameter; return true in this case.

    // If key is not equal to any key currently in the map and if the

    // key/value pair can be added to the map, then do so and return true.

    // Otherwise, make no change to the map and return false (indicating

    // that the key is not already in the map and the map has a fixed

    // capacity and is full).

    bool erase(const KeyType& key)

designate pointer “found” to saerching for key

if found

if DataMember has no pre or next

delete found and set head to null

else if found is headptr

set next’s Data member’s previous to null

head is set to next

delete found

else if found is tailptr

set rev’s next to null

delete found

else found is in the middle

set prev’s next to found’s next

set next’s prev to found’s prev

delete found

return true

return false

    // If key is equal to a key currently in the map, remove the key/value

    // pair with that key from the map and return true.  Otherwise, make

    // no change to the map and return false.

    bool contains(const KeyType& key) const

check if search fun. is null

otherwise return false

    // Return true if key is equal to a key currently in the map, otherwise

    // false.

    bool get(const KeyType& key, ValueType& value)const

create ptr “daSearch” pointing to head

loop runs through daSaerch

if key in daSearch equals to input key

set value to daSearch key’s value and return true

otherwise, return false

    // If key is equal to a key currently in the map, set value to the

    // value in the map that that key maps to, and return true.  Otherwise,

    // make no change to the value parameter of this function and return

    // false.

    bool get(int i, KeyType& key, ValueType& value)const

if i is within range

create ptr daSearch

have j run through list for i-times

set deSearch’s key and value to the input key/values and return true

otherwise, return false

    // If 0 <= i < size(), copy into the key and value parameters the

    // key and value of one of the key/value pairs in the map and return

    // true.  Otherwise, leave the key and value parameters unchanged and

    // return false.  (See below for details about this function.)

    void swap(Map& other)

create a temp. ptr

swap the ptrs using temp

create temp. int

swap current amoutns using temp

    // Exchange the contents of this map with the other one.

    void dump() const;

    //The intent of this function is that for your own testing purposes, you can call it to print information about the map; we will never call it. You do not have to add this function if you don't want to, but if you do add it, it must not make any changes to the map; if we were to replace your implementation of this function with one that simply returned immediately, your code must still work correctly. The dump function must not write to cout, but it's allowed to write to cerr.

    ~Map()

run through headptr’s list

create ptr “curr” which hold headPtr

incrment headPtr and delete curr

delete headPtr

    //When a Map is destroyed, the nodes in the linked list must be deallocated.

private:

    class DataMember

    {

    public:

        void setKey(KeyType key);

        ValueType getValue()const;

        void setValue(ValueType value);

        KeyType getKey()const;

        DataMember\* nextPtr;

        DataMember\* prevPtr;

    private:

        KeyType key;

        ValueType value;

    };

    DataMember\* search(KeyType key) const;

    DataMember\* checkIfCan(Map p, DataMember\* ptr);

    int currentAmount;

    DataMember\* headPtr;

};

bool combine(const Map& m1, const Map& m2, Map& result)

bool to keep track of scenario

for loop runs through result’s list

get key/value pairs at i

erase this key/value pair from result

for loop runs through m1

get key/value pairs at i

insert key/value pair into result

for loop runs through m2

get m2’s key/value pair at i

if result has m2 key

get result’s value

if result’s value is equal to m2’s value

scenario is false and erase key/value pair

else continue;

insert key/value pair into result

return scenario

//    When this function returns, result must consist of pairs determined by these rules:

//

//    If a key appears in exactly one of m1 and m2, then result must contain a pair consisting of that key and its corresponding value.

//    If a key appears in both m1 and m2, with the same corresponding value in both, then result must contain a pair with that key and value.

void subtract(const Map& m1, const Map& m2, Map& result)

for loop runs through result

get result’s i’th pair and erase it

for loop runs through  m1

get m1’s i’th pair and insert into result

for loop runs through m2

get m2’s pair

if result contains nay keys of m2, delete from result

//    When this function returns, result must contain a copy of all the pairs in m1 whose keys don't appear in m2; it must not contain any other pairs. (You must not assume result is empty when it is passed in to this function; it may not be.)

1. List of Test-Cases:

//tests for map: strings/doubles

Map m;

//for an empty map:

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

assert(m.empty());

assert(!m.erase("Ricky"));

//for copy constructors

Map n(m);

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

assert(n.empty());

assert(!n.erase("Ricky"));

m.insert("Hey", 43); //comment out and assert(n.size(2))

m.insert("noo", 3); //comment out and assert(n.size(1))

m.insert("yes", 3); //comment out and assert(n.size(0))

Map d(m);

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

assert(!d.empty());

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

//for assignment operator

Map z = d;

z.insert("mamamia", 59);

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

assert(!z.empty());

Map b;

b.insert("yiss", 75); //checking if nonempty maps function

b.insert("nananana", 83);

b = z;

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

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

//for insert

Map i;

i.insert("yoyoma", 23);

i.insert("", 543); //check if empty strings accepted

assert(i.contains(""));

//for update

assert(z.update("mamamia", 40));

assert(!z.update("aybaby", 40)); //checking if non-existing node returns false

//for insertOrUpdate

assert(z.insertOrUpdate("usuck", 4839));

assert(z.contains("usuck"));

assert(z.insertOrUpdate("aybaby", 4839));

assert(z.contains("aybaby")); //no way for insertOrUpdate to be false bc no max amt of nodes

//for erase

assert(i.erase("")); //checking for empty strings

assert(!(i.contains("")));

assert(z.erase("aybaby"));

assert(!z.contains("aybaby"));

assert(!z.erase("truest"));

assert(z.erase("Hey")); //check if head is deleted

assert(z.erase("usuck")); //check if tail is deleted

assert(z.insert("bobo", 69));

assert(z.erase("mamamia"));

//for contains

assert(!z.contains("mamamia"));

assert(z.contains("bobo"));

assert(!z.erase("mamamia"));

assert(z.erase("bobo"));

assert(z.erase("noo"));

assert(!z.contains("bobo")); //when there is no list

//for get version 1.0

z.insert("Hey", 43);

z.insert("noo", 3);

z.insert("yes", 3);

ValueType v;

z.get("yes", v);

assert(v == 3);

assert(!z.get("brehbreh", v));

//for get version 2.0

KeyType k;

assert(!z.get(10, k, v));

assert(!z.get(-5, k, v));

assert(z.get(0, k, v)); //check for first node

assert(k == "Hey");

assert(v == 43);

assert(z.get(1, k, v)); //check for middle node

assert(!(k == "Hey"));

assert(!(v == 43));

assert(k == "noo");

assert(v == 3);

assert(z.get(2, k, v)); //check for last node

assert(k == "yes");

assert(v == 3);

//for swap

z.swap(d);

assert(!d.get(10, k, v));

assert(!d.get(-5, k, v));

assert(d.get(0, k, v)); //check for first node

assert(k == "Hey");

assert(v == 43);

assert(d.get(1, k, v)); //check for middle node

assert(!(k == "Hey"));

assert(!(v == 43));

assert(k == "noo");

assert(v == 3);

assert(d.get(2, k, v)); //check for last node

assert(k == "yes");

assert(v == 3);

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

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

//for combine

d.insert("yum", 5438.5);

z.insert("yum", 38);

combine(d, z, m);

assert(!m.contains("yum"));

assert(m.contains("noo")); //check for head

assert(m.contains("Hey")); //check for middle

assert(m.contains("yes")); //check for end

//for subtract

subtract(d, z, m);

assert(m.contains("yes"));

assert(!(m.contains("Hey")));

subtract(z, d, m);

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

//tests for map: ints/strings

Map m;

//for an empty map:

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

assert(m.empty());

assert(!m.erase(8));

//for copy constructors

Map n(m);

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

assert(n.empty());

assert(!n.erase(8));

m.insert(15, "Hey"); //comment out and assert(n.size(2))

m.insert(78, "noo"); //comment out and assert(n.size(1))

m.insert(62, "yes"); //comment out and assert(n.size(0))

Map d(m);

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

assert(!d.empty());

assert(d.erase(62));

//for assignment operator

Map z = d;

z.insert(59, "mamamia");

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

assert(!z.empty());

//for insert

Map i;

i.insert(1, "yoyoma");

i.insert(7, "");

assert(i.contains(7));

//for update

assert(z.update(59, "oyo"));

assert(!z.update(6, "jkdslfa"));

//for insertOrUpdate

assert(z.insertOrUpdate(91, "usuck"));

assert(z.contains(91));

assert(z.insertOrUpdate(2, "aybaby"));

assert(z.contains(2));

//for erase

assert(i.erase(7));

assert(!(i.contains(7)));

assert(z.erase(2));

assert(!z.contains(2));

assert(!z.erase(3));

assert(z.erase(15)); //check if head is deleted

assert(z.erase(91)); //check if tail is deleted

assert(z.insert(4, "bobo"));

assert(z.erase(59)); //look again

//for contains

assert(!z.contains(59));

assert(z.contains(4));

assert(!z.erase(59));

assert(z.erase(4));

assert(z.erase(78));

assert(!z.contains(4)); //when there is no list

//for get version 1.0

z.insert(15, "Hey");

z.insert(78, "noo");

z.insert(62, "yes");

ValueType v;

z.get(62, v);

assert(v == "yes");

assert(!z.get(5, v));

//for get version 2.0

KeyType k;

assert(!z.get(10, k, v));

assert(!z.get(-5, k, v));

assert(z.get(0, k, v)); //check for first node

assert(k == 15);

assert(v == "Hey");

assert(z.get(1, k, v)); //check for middle node

assert(!(k == 15));

assert(!(v == "Hey"));

assert(k == 78);

assert(v == "noo");

assert(z.get(2, k, v)); //check for last node

assert(k == 62);

assert(v == "yes");

//for swap

z.swap(d);

assert(!d.get(10, k, v));

assert(!d.get(-5, k, v));

assert(d.get(0, k, v)); //check for first node

assert(k == 15);

assert(v == "Hey");

assert(d.get(1, k, v)); //check for middle node

assert(!(k == 15));

assert(!(v == "Hey"));

assert(k == 78);

assert(v == "noo");

assert(d.get(2, k, v)); //check for last node

assert(k == 62);

assert(v == "yes");

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

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

//for combine

d.insert(9, "yum");

z.insert(9, "yumdos");

combine(d, z, m);

assert(!m.contains(9));

assert(m.contains(78)); //check for head

assert(m.contains(15)); //check for middle

assert(m.contains(62)); //check for end

//for subtract

subtract(d, z, m);

assert(m.contains(62));

assert(!(m.contains(15)));

subtract(z, d, m);

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