Report

CS32 Project 2

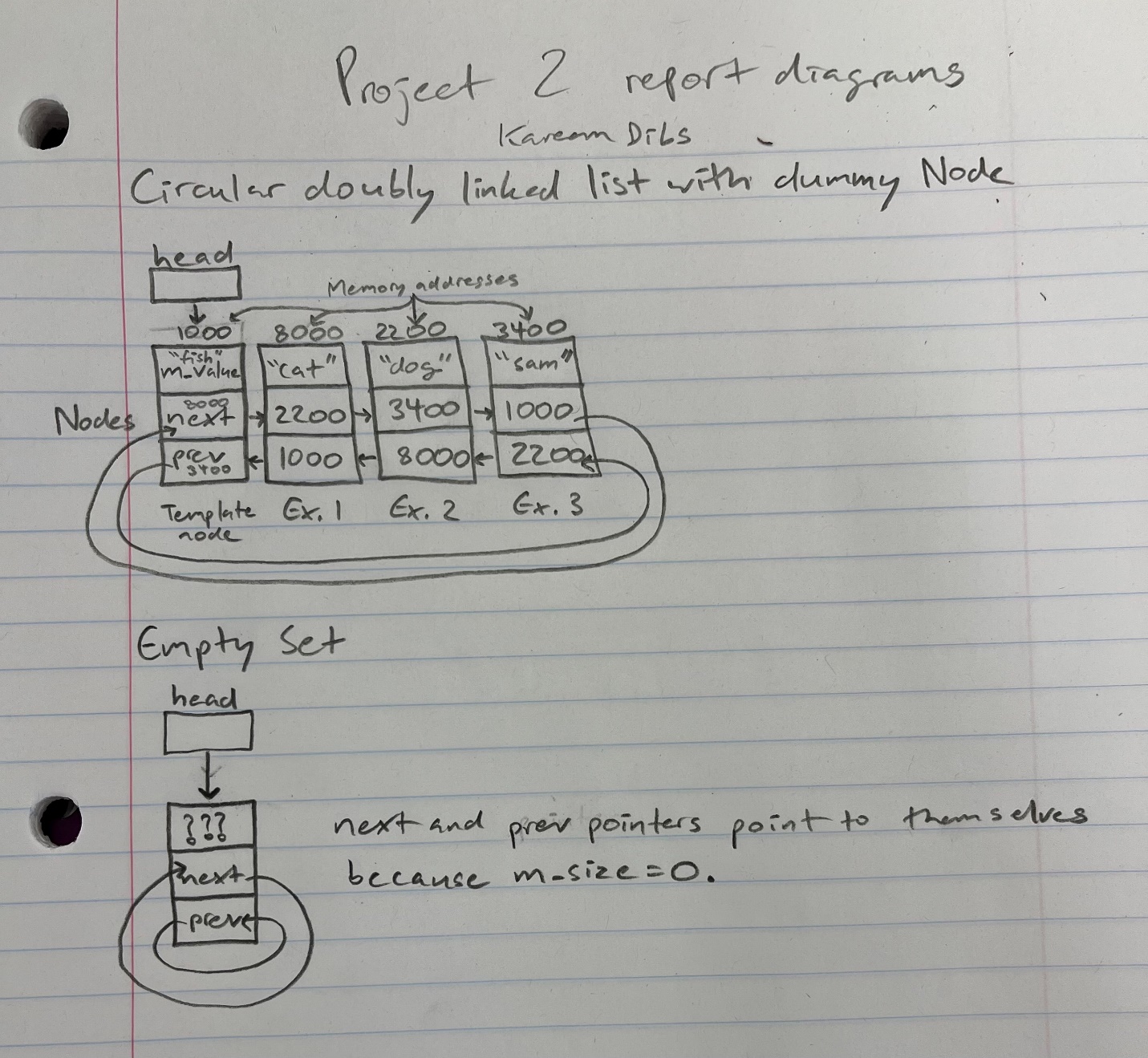
Kareem Dibs

1. A description of the design of your doubly-linked list implementation. (A couple of sentences will probably suffice, perhaps with a picture of a typical Set and an empty Set. Is the list circular? Does it have a dummy node? What's in your list nodes? Are they in any particular order?)

In my Set class, I utilize circular doubly linked lists and a dummy node that is pointed to by the head Node. Each node contains one ItemType m\_value, which allows that node to store one value of ItemType in it. Additionally, each node contains two pointers, next and prev, with next pointing to the next Node and prev pointing to the previous Node. When the Set’s m\_size = 0, meaning the Set is empty, the dummy Node’s next and prev pointers point to itself. By using a dummy node instead of a head node, there is no alternative procedure for adding a new Node to the top of the list, as the dummy Node is a regular Node just like all of the other nodes. When I am inserting new Nodes into the linked list while using the insert function, I insert a new Node at the end of the linked list.

Diagrams on next page.

DIAGRAMS: Circular Doubly Linked List with Dummy Node & Empty Set.



1. pseudocode for non-trivial algorithms.

**Copy constructor**

Set(const Set& src) {

Assign src’s m\_size to this Set’s size

Allocate new space in memory for this Set’s linked list

Make dummy node’s next and prev pointers point to dummy node

Create a temp ptr pointing to newly added linked list

Create a temp ptr pointing to src’s lists

repeatedly:

create new valPtr to assign src to this

assign src’s m\_value to the new Set’s m\_value

make newly added node’s next point to dummy node

make newly added node’s prev point to previous node

connect prev node to new node

complete circle by connecting first and last nodes

}

**Destructor**

~Set() {

repeatedly:

while Set’s m\_size != 0, meaning it’s not empty

create temp ptr to point to the first item to be deleted

adjust ptr so head ptr pointers to the next node

if only one m\_value item in linked list, point to head

complete circle by making temp->next->prev = head

make soon-to-be-deleted temp node’s pointers equal to nullptr so that they cannot be

incorrectly accessed

delete current m\_value item

decrement m\_size

delete dummy node

}

**Insert**

bool insert(const ItemType& value) {

return false if value already exists in Set

if Set is empty

create new Node p to insert new value after dummy Node

set p->m\_value to value

connect nodes p and head

increment size

return true

create new Node q to insert new value in the middle of Set, q = head->next

repeatedly (while q != head): // q = head would complete the circle

if m\_value > value

create new Node z to place in Set

set z->m\_value to value

connect nodes q and z

increment size

return true

increment q to q->next if q->m\_value is not greater than value

create new Node end to place new Node at the end of the Set

set end->m\_value to value

connect nodes end and head

increment size

return true

}

**Erase**

bool erase(const ItemType& value) {

if list doesn’t contain value, return false

utilizing next, traverse to position in linked list where value is located

make temp pointer point to location of value in linked list

delete value

make soon-to-be-deleted node temp's pointers equal to nullptr

delete temp

set deleted node temp to nullptr

decrement size

return true

}

**Get**

bool get(int i, ItemType& value) const {

if pos > 0 or pos >= m\_size

return false

create temp pointer to locate position of value

repeatedly: // iterate using next

stop iterating when temp node pointer points to node at position of value

assign that node’s m\_value to the value in the parameter

return true

}

**Swap**

void swap(Set& other) {

Swap sizes using tempSize variable

Swap head pointers to the linked list using swapHeadNodes Node

}

**Contains**

bool contains(const ItemType& value) const

create new node pointing to head->next

repeatedly:

iterate through linked list until circle is complete

check if linked list already contains value. if it does, return true.

return false if value is not found in linked list

}

**Unite**

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

create a new empty set called tempSet

repeatedly:

insert s1’s list into tempSet // utilize insert function to avoid duplicates

repeatedly:

insert s2’s list into tempSet // utilize insert function to avoid duplicates

assign tempSet to result

}

**butNot**

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

create a new empty set called tempSet

use unite function to unite Set s1 and Set s2 into tempSet

repeatedly:

erase items of s2 in tempSet

assign tempSet to result

}

Test Cases on next page.

1. Test Cases

#include "Set.h"

#include <iostream>

#include <cassert>

using namespace std;

int main() {

Set a; //Check if constructor works

Set b;

Set c;

c = b; // Check assignment operator

assert(a.size() == 0); // Check that initial size is zero

assert(a.size() == b.size());

assert(b.size() == c.size());

assert(a.empty() && b.empty() && c.empty()); // Check that empty function works correctly

a.insert("insane"); finished

a.insert("wonderful"); smart

a.insert("awesome"); talent

a.insert("awesome"); talent

assert(a.size() == 3); // Check that insert function only works if value is not already present

b.insert("insane");

b.insert("wonderful");

b.insert("awesome");

b.insert("excellent"); formidable

c.insert("chad"); zap

Set d(c); // Check the copy constructor with only one node

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

Set e(b); // Check that copy constructor works with multiple nodes

assert(e.size() == b.size());

assert(c.size() == 1); // Check that INSERT increases the size

assert(!c.empty());

unite(a, b, c);

assert(c.contains("insane") && c.contains("wonderful") && c.contains("awesome") && c.contains("excellent")); // Check that Unite function works properly

unite(a, a, a);

assert(c.contains("insane") && c.contains("wonderful") && c.contains("awesome") && c.size() == 3); // Check last aliasing case

unite(a, b, a);

assert(c.contains("insane") && c.contains("wonderful") && c.contains("awesome") && c.size() == 3); // Check alternate aliasing case

unite(a, a, c);

assert(c.contains("insane") && c.contains("wonderful") && c.contains("awesome") && c.size() == 3); // Check aliasing case

butNot(a, b, c);

assert(c.size() == 0); // Check that butNot works properly

a.insert("excellent");

a.insert("lovely"); fiction

butNot(a, b, c);

assert(c.size() == 1 && c.contains("lovely")); // Check that butNot works properly

butNot(b, b, b);

assert(b.size() == 0); // Check last aliasing case

butNot(a, a, c);

assert(c.size() == 0); // Check aliasing case

assert(!a.erase("basketball")); // Check that erasing something that is not present returns false

assert(a.erase("lovely")); //

Check that erasing something that is already present returns true

assert(a.size() == 4); // Check that Erase function decreases size

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

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

a.insert("insane");

a.insert("wonderful");

a.insert("awesome");

b.insert("insane");

// Check that two reciprocal swaps cancel out

a.swap(b);

b.swap(a);

assert(a.contains("insane") && a.contains("wonderful") && a.contains("awesome")); // Check that CONTAINS works properly

assert(!c.contains("insane"));

a.dump();

ItemType newItem;

a.get(0, newItem);

assert(newItem == "insane");

//Check that GET works properly

a.get(1, newItem);

assert(newItem == "excellent");

a.get(2, item);

assert(newItem == "wonderful");

b.dump();

a.swap(b); // Check that Swap function works properly

assert(a.size() == 1 && a.contains("insane") && b.size() == 4 && b.contains("insane") && b.contains("wonderful") && b.contains("awesome"));

b.get(0, newItem);

assert(newItem == "insane"); // Check that Swap function works properly

b.get(1, newItem);

assert(newItem == "excellent");

b.get(2, newItem);

assert(newItem == "wonderful");

// Check empty Set

Set empty;

Set not\_empty(a);

assert(empty.empty());

assert(!not\_empty.empty());

empty.swap(not\_empty); // Check Swap function when swapping an empty set

assert(!empty.empty());

assert(not\_empty.empty());

cout << "Passed All Tests!" << endl << endl;

}