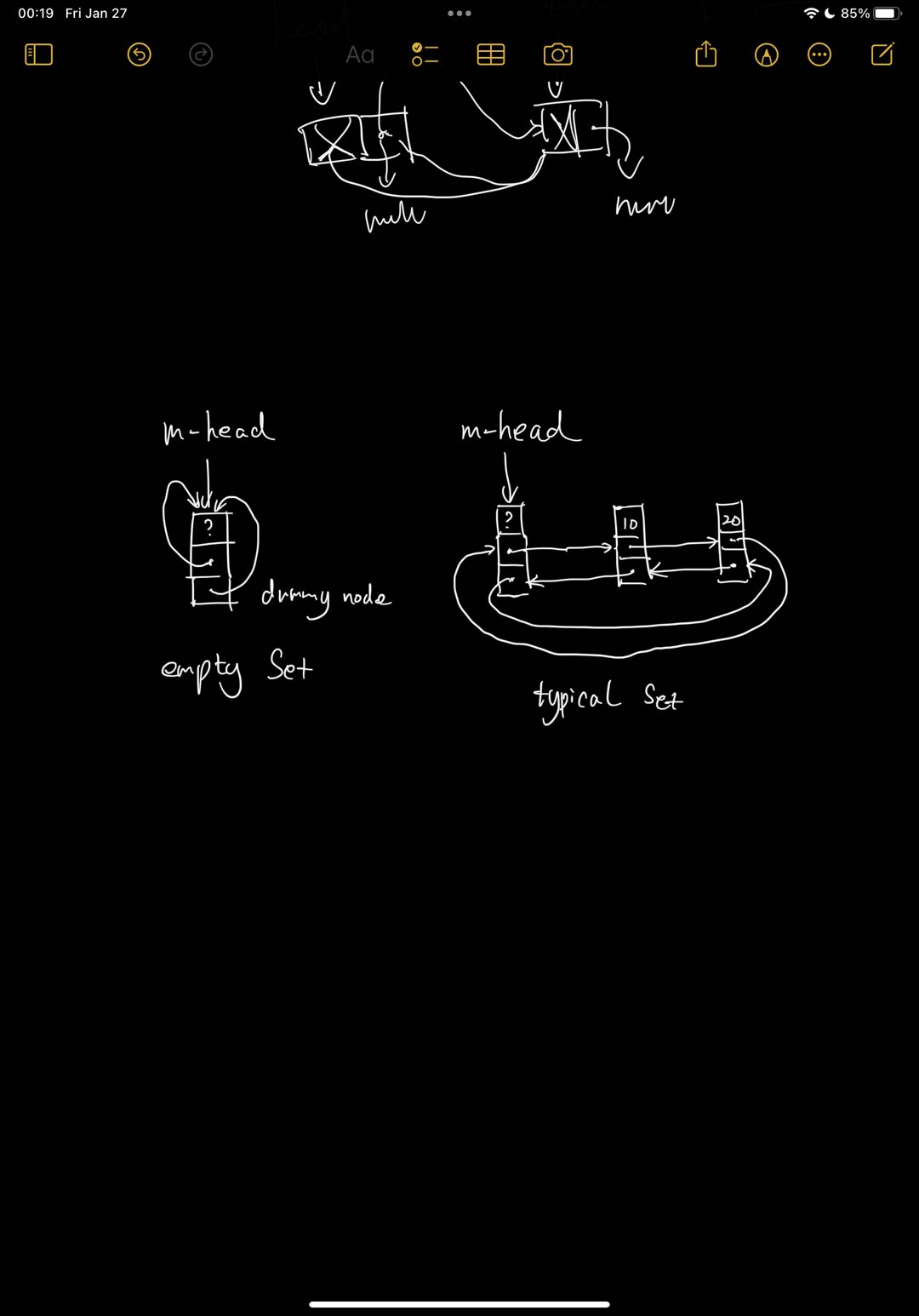
**CS32 Project 2 Report**

Xuanhe Zhang UID: 706078980 Jan 29, 2023

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?)**



My doubly-linked list has one *m\_head* pointer which point to the dummy node.

It is circular. The dummy node’s *prev* and *next* pointers are pointed to itself in the beginning.

The list nodes contain two pointers called *prev* and *next* which pointing to the previous node and the next node. It also has a ItemType *data* to hold value.

It is maintaining an **ascending** and **non-repeated** order.

1. **pseudocode for non-trivial algorithms (e.g., butNot).**

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

deep copy each node from *s1* to *result*;

traverse *s2*’s nodes:

insert *value* of each node to the *result*;

}

}

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

deep copy each node from s1 to result;

traverse *s2*’s nodes:

erase *value* of each node in the *result*;

}

Set::Node\* Set::findFirstAtLeast(const ItemType& *value*) const {

if *dummy head* point to itself:

return *dummy head*

else

traverse nodes that less than *value* and is not *dummy head*:

return

the next node which equal or larger than *value* or *dummy head*

}

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

{

call findFirstAtLeast(*value*) to get *first node* at least *value*

if first node’s data equal to *value*:

return false

create a *new node* holds *value*

let *previous node*’s next pointer to *new node*

let *first node*’s previous pointer to *new node*

let new node’s previous and next to *previous node* and *first node*

*size* add 1

return true

}

bool Set::erase(const ItemType& value)

{

call findFirstAtLeast(value) to get *first node* at least value

if *first node*’s data not equal to value:

return false

let *previous node*’s next pointer to first node’s *next node*

let *next node*’s previous pointer to *previous node*

delete *first node*

*size* minus 1

return true

}

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

{

if *i* less than 0 or larger than set’s *size*, return false

traverse *i* times from *first non-dummy node* to find *correct node*

set *correct node*’s data to value

return true

}

1. **A list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. For example, here's the beginning of a presentation in the form of code:**

Set s1; // default constructor

assert(s1.empty()); // test empty function

ItemType x = "arepa";

assert(!s1.get(42, x) && x == "arepa"); // x unchanged by get failure

s1.insert("chapati"); // test insert function

assert(s1.size() == 1); // test size function

assert(s1.get(0, x) && x == "chapati"); //test get function

s1.insert("ahapati");

assert(s1.size() == 2); // test size function works when insert

assert(s1.get(0, x) && x == "ahapati"); //test get function

Set s2;

s2.insert("zill");

s2.insert("fill");

s2.insert("bill");

assert(s2.contains("bill")); // test contains function

s2.erase("bill"); // test erase function

assert(s2.contains("fill"));

assert(!s2.insert("fill"));

// test data structures will not accpet duplicated data

s2.insert("cool");

assert(!s2.contains("bill")); // test erase function

assert(s2.contains("fill")); // test erase function

assert(!s2.insert("fill"));

s2.insert("");

assert(s2.get(0, x) && x == ""); // test empty string works

s1.swap(s2); // test swap function

assert(s1.contains("") && !s2.contains("cool") && s2.contains("chapati")); // test swap function swap two sets'data correctly

assert(s1.size() == 4 && s2.size() == 2);

// test swap function swap two set's size correctly

assert(!s1.erase("nill")); // test erase function for inexist data

assert(s1.erase("fill")); // test erase function

assert(s1.size() == 3); // test size function works when erase data

assert(!s1.contains("fill")); // test swap function works

assert(!s1.contains("chapati")); //test swap function works

Set full;

for (int i = 0; i < 160; i++) {

// test double linked list works under a lot of data

full.insert(std::to\_string(i));

}

assert(full.size() == 160); // test size function under a lot of data

assert(full.contains("4")); // test contains function under a lot of data

assert(full.erase("4")); // test erase function under a lot of data

assert(!full.contains("4"));

assert(full.insert("K")); // test insert function under a lot of data

assert(full.contains("K"));

assert(full.size() == 160);

ItemType value;

ItemType& v = value;

assert(!full.get(160, v)); // test get function under a lot of data

assert(full.get(159, v));

assert(v == "K");

assert(full.get(0, v));

assert(v == "0");

Set emptyS;

Set s3;

Set s4;

Set uniteS;

Set butnotS;

assert(s3.insert("c") && s3.insert("a") && s3.insert("b"));

assert(s4.insert("b") && s4.insert("d"));

unite(s3, s4, uniteS);

assert(uniteS.contains("a") && uniteS.contains("b") && uniteS.contains("c") && uniteS.contains("d") && uniteS.size() == 4);

// test unite function works

unite(uniteS, emptyS, butnotS);

assert(butnotS.contains("a") && butnotS.contains("b") && butnotS.contains("c") && butnotS.contains("d") && butnotS.size() == 4);

// test unite function unite a empty set works

butNot(s3, s4, butnotS);

assert(butnotS.size() == 2 && butnotS.contains("a") && butnotS.contains("c") && !butnotS.contains("b") && !butnotS.contains("d"));

// test butNot function works when it is not empty

butNot(s4, emptyS, butnotS);

assert(butnotS.size() == 2 && !butnotS.contains("a") && !butnotS.contains("c") && butnotS.contains("b") && butnotS.contains("d"));

// test butNot function works when one set is empty