# Design

The list is a circular, doubly-linked list with a dummy node. Each node has a pointer to the previous node, ‘prev’, a pointer to the next node, ‘next’, and a value of type ItemType, ‘item’. The dummy node’s ‘next’ pointer points to the first node in the list, and the ‘prev’ pointer points to the last node in the list, thereby making it a circular list. In an empty list, the dummy node’s ‘prev’ and ‘next’ pointers point to itself. The dummy node’s ‘item’ value is left uninitialized as it is never accessed. The nodes are not in any particular order.

# Pseudocode

## Copy constructor

create empty set;

traverse through all the nodes of the other set {

insert value of node into new node in current set;

}

## Destructor

traverse through all nodes in the set using p {

temp pointer points to current node;

point p to next node;

delete temp;

}

delete p;

## Assignment operator

if both sets are the same

return set;

delete all nodes except dummy node; //effectively empty set

traverse through all the nodes of the other set {

insert value of node into new node in current set;

}

return set;

## insert

if set already contains value

return false;

create new node;

assign value to new node;

make new node point to previously last item in set and dummy node;

size++;

make previously last item in set point to new node;

make dummy node point to new node;

return true;

## erase

point p to node in set containing value;

if node was not found

return false;

make nodes before and after p point to each other;

delete p;

size--;

return true;

## contains

traverse through all nodes in set {

if item in current node is equal to value

return true;

}

return false;

## get

if pos is out of bounds

return false;

traverse through all nodes in set using p {

count = 0;

traverse through all nodes in set using q {

if item in p is greater than item in q

count++;

if count is equal to pos {

set value equal to item in p;

return true;

}

return false;

## swap

swap pointers to dummy nodes using temp pointer;

swap sizes using temp size;

## unite

create temp empty set;

traverse through s1 {

get item from s1;

insert item into temp set;

}

traverse through s2 {

get item from s2;

insert item into temp set;

}

set result equal to temp set;

## subtract

create temp empty set;

traverse through s1 {

get item from s1;

insert item into temp set;

}

traverse through s2 {

get item from s2;

delete item (if contained) from temp set;

}

set result equal to temp set;

# Test Cases

## Tests given by spec

### Test 1

#include "Set.h"

#include <type\_traits>

#define CHECKTYPE(f, t) { auto p = static\_cast<t>(f); (void)p; }

static\_assert(std::is\_default\_constructible<Set>::value,

"Set must be default-constructible.");

static\_assert(std::is\_copy\_constructible<Set>::value,

"Set must be copy-constructible.");

static\_assert(std::is\_copy\_assignable<Set>::value,

"Set must be assignable.");

void thisFunctionWillNeverBeCalled()

{

CHECKTYPE(&Set::empty, bool (Set::\*)() const);

CHECKTYPE(&Set::size, int (Set::\*)() const);

CHECKTYPE(&Set::insert, bool (Set::\*)(const ItemType&));

CHECKTYPE(&Set::erase, bool (Set::\*)(const ItemType&));

CHECKTYPE(&Set::contains, bool (Set::\*)(const ItemType&) const);

CHECKTYPE(&Set::get, bool (Set::\*)(int, ItemType&) const);

CHECKTYPE(&Set::swap, void (Set::\*)(Set&));

CHECKTYPE(unite, void (\*)(const Set&, const Set&, Set&));

CHECKTYPE(subtract, void (\*)(const Set&, const Set&, Set&));

}

int main()

{}

### Test 2 (using ItemType = std::string;)

#include "Set.h"

#include <iostream>

#include <cassert>

using namespace std;

void test()

{

Set ss;

assert(ss.insert("roti"));

assert(ss.insert("pita"));

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

assert(ss.contains("pita"));

ItemType x = "laobing";

assert(ss.get(0, x) && x == "pita");

assert(ss.get(1, x) && x == "roti");

}

int main()

{

test();

cout << "Passed all tests" << endl;

}

### Test 3 (using ItemType = unsigned long;)

#include "Set.h"

#include <iostream>

#include <cassert>

using namespace std;

void test()

{

Set uls;

assert(uls.insert(10));

assert(uls.insert(20));

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

assert(uls.contains(20));

ItemType x = 30;

assert(uls.get(0, x) && x == 10);

assert(uls.get(1, x) && x == 20);

}

int main()

{

test();

cout << "Passed all tests" << endl;

}

## Tests were performed on sets of strings (using ItemType = std::string;)

Set a;

*//testing default constructor*

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

*//testing size function on empty set*

assert(a.empty());

*//testing empty function on empty set*

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

*//testing insert function*

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

*//testing size function on non empty set*

assert(!a.empty());

*//testing empty function on non empty set*

assert(!a.erase("b"));

*//testing erase to ensure values that don’t match b don’t get erased*

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

*//testing insert not adding an already present value*

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

*//testing erase to make sure present values are erased*

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

assert(a.contains("a"));

*//testing contains function*

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

*//testing contains function for nonexistent value*

a.insert("b");

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

*//making sure contains works for b that was added*

Set b(a);

*//testing copy constructor*

assert(b.contains("a") && b.contains("b") && b.size() == 2);

*//making sure a was copied into b*

Set c;

Set d(c);

*//testing copy constructor for empty set*

assert(!d.size());

*//making sure d is empty*

assert(d.empty());

string ab = "";

assert(a.get(1, ab));

*//testing get function*

assert(ab == "b");

*//b is greater than one value*

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

*//a is smallest value*

assert(!d.get(0, ab));

*//testing get return value (should return false for empty set)*

assert(!a.get(2, ab));

*//testing get return value for out of bounds case (2 > m\_size – 1, should return false)*

Set e;

swap(e, d);

*//testing swap with two empty sets*

swap(a, d);

*//testing swap function with empty set and non empty set*

assert(a.size() == 0 && d.size() == 2 && d.contains("a") && d.contains("b"));

*//making sure values were swapped*

e = c = d;

*//testing assignment operator*

assert(c.size() == 2 && c.contains("b"));

*//ensuring values are consistent between c and d*

assert(e.size() == 2 && e.contains("b"));

*//ensuring values are consistent between e and c*

a.insert("f");

a.insert("as");

unite(a, c, d);

*//testing unite function with two non empty sets*

assert(d.size() == 4 && d.contains("as"));

*//ensuring result set is accurate*

subtract(d, a, c);

*//testing subtract function with two non empty sets*

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

*//ensuring that the two values leftover are consistent with the requirements for subtract*

Set f;

a = b = f;

*//check assignment operator with empty sets*

assert(a.empty() && b.empty());

*//check to make sure they are empty*

unite(a, b, f);

*//testing unite function with empty sets*

assert(f.empty());

subtract(a, b, f);

*//testing subtract function with empty sets*

assert(f.empty());

a.insert("a");

b.insert("b");

unite(a, b, a);

*//testing unite with aliasing, ie when result set == s1 or s2*

assert(a.size() == 2 && a.contains("b"));

*//making sure the result set was correctly modified*

subtract(b, a, f);

*//testing subtract when set2 is bigger than set1 and has all of set1’s values*

assert(f.empty());

a = f;

f.insert("a");

f.insert("b");

a.insert("c");

a.insert("d");

a.insert("e");

subtract(a, f, f);

*//Check if subtract function works if result and arguments are the same*

assert(f.size() == 3 && f.contains("c"));

*//since a and f didn’t have any values in common, f*

assert(f.contains("d") && f.contains("e"));

*//should contain everything a has*