**Project 2 Test Data**

When the following program is compiled along with Set.cpp, it can be run with input 1 through 74 to test your solution. (This program uses some features of C++ we're not covering, but that's irrelevant to you.) You got 1 point for turning in something non-trivial, and each test case was worth 1 point.

To be able to keep track of how many nodes you created and deleted, we devised a simple class that acted like an int, but kept track of how many objects of that class were in existence at any time. We called this class OurType. We adjusted your ItemType typedef so that your Set stored objects of this type (and we inserted #include "OurType.h" in your Set.h file). (This test harness uses some C++ features we're not covering, but that should be irrelevant to you.)

=========================== OurType.h ==============================

#ifndef OURTYPE\_INCLUDED

#define OURTYPE\_INCLUDED

struct OurType

{

explicit OurType(int v = 0) : m\_value(v) { m\_count++; }

OurType(const OurType& other) : m\_value(other.m\_value)

{ m\_count++; m\_asstcopycount++; }

~OurType() { m\_count--; }

OurType& operator=(const OurType& rhs)

{ m\_value = rhs.m\_value; m\_asstcopycount++; return \*this; }

int m\_value;

static int m\_count;

static int m\_asstcopycount;

};

inline

bool operator==(const OurType& lhs, const OurType& rhs)

{ return lhs.m\_value == rhs.m\_value; }

inline

bool operator<(const OurType& lhs, const OurType& rhs)

{ return lhs.m\_value < rhs.m\_value; }

inline

bool operator!=(const OurType& lhs, const OurType& rhs)

{ return ! (lhs == rhs); }

inline

bool operator<=(const OurType& lhs, const OurType& rhs)

{ return ! (rhs < lhs); }

inline

bool operator>=(const OurType& lhs, const OurType& rhs)

{ return ! (lhs < rhs); }

inline

bool operator>(const OurType& lhs, const OurType& rhs)

{ return rhs < lhs; }

inline

bool operator==(const OurType& lhs, int rhs)

{ return lhs.m\_value == rhs; }

inline

bool operator!=(const OurType& lhs, int rhs)

{ return ! (lhs == rhs); }

#include <iostream>

inline

std::ostream& operator<<(std::ostream& lhs, const OurType& rhs)

{ return lhs << rhs.m\_value; }

#endif // OURTYPE\_INCLUDED

=========================== tester.cpp ==============================

#include "OurType.h"

#include "Set.h"

#include <iostream>

#include <cstdlib>

#include <cassert>

using namespace std;

int OurType::m\_count = 0;

int OurType::m\_asstcopycount = 0;

inline int itemcount()

{

return OurType::m\_count;

}

inline int nasstcopys()

{

return OurType::m\_asstcopycount;

}

bool emptytype(bool (Set::\*)() const) { return true; }

bool emptytype(bool (Set::\*)()) { return false; }

bool emptytype(...) { return false; }

bool sizetype(int (Set::\*)() const) { return true; }

bool sizetype(int (Set::\*)()) { return false; }

bool sizetype(...) { return false; }

bool containstype(bool (Set::\*)(const ItemType&) const) { return true; }

bool containstype(bool (Set::\*)(const ItemType&)) { return false; }

bool containstype(...) { return false; }

bool gettype(bool (Set::\*)(int, ItemType&) const) { return true; }

bool gettype(bool (Set::\*)(int, ItemType&)) { return false; }

bool gettype(...) { return false; }

ItemType SOMEVAL = ItemType(123);

ItemType DEFAULT = ItemType(0);

ItemType ARRAY[5] = {

ItemType(10), ItemType(20), ItemType(30), ItemType(40), ItemType(50)

};

bool isPermutation(ItemType a[], const ItemType b[], int n)

{

for (size\_t k = 0; k < n; k++)

{

size\_t j;

for (j = k; j < n; j++)

{

if (b[k] == a[j])

{

ItemType t = a[k];

a[k] = a[j];

a[j] = t;

break;

}

}

if (j == n)

return false;

}

return true;

}

void testone(int n)

{

Set s;

switch (n)

{

default: {

cout << "Bad test number" << endl;

} break; case 1: {

assert(emptytype(&Set::empty));

} break; case 2: {

assert(sizetype(&Set::size));

} break; case 3: {

assert(containstype(&Set::contains));

} break; case 4: {

assert(gettype(&Set::get));

} break; case 5: {

assert(s.empty());

} break; case 6: {

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

} break; case 7: {

assert(!s.erase(DEFAULT) && s.size() == 0);

} break; case 8: {

assert(!s.contains(DEFAULT));

} break; case 9: {

ItemType x = SOMEVAL;

assert(!s.get(0, x));

} break; case 10: {

ItemType x = SOMEVAL;

s.get(0, x);

assert(x == SOMEVAL);

} break; case 11: {

assert(s.insert(SOMEVAL));

} break; case 12: {

s.insert(SOMEVAL);

assert(!s.empty());

} break; case 13: {

s.insert(SOMEVAL);

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

} break; case 14: {

s.insert(SOMEVAL);

assert(s.contains(SOMEVAL));

} break; case 15: {

s.insert(SOMEVAL);

ItemType x = DEFAULT;

assert(s.get(0, x));

} break; case 16: {

s.insert(SOMEVAL);

ItemType x = DEFAULT;

s.get(0, x);

assert(x == SOMEVAL);

} break; case 17: {

assert(s.insert(SOMEVAL) && !s.insert(SOMEVAL));

} break; case 18: {

s.insert(SOMEVAL);

s.insert(SOMEVAL);

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

} break; case 19: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

assert(!s.empty() && s.size() == 2);

} break; case 20: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

ItemType x = SOMEVAL;

s.get(0,x);

assert(x == ARRAY[0] || x == ARRAY[1]);

} break; case 21: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

ItemType x = SOMEVAL;

s.get(0,x);

ItemType y = SOMEVAL;

s.get(0,y);

assert(x == y);

} break; case 22: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

ItemType x = SOMEVAL;

s.get(0,x);

ItemType y = SOMEVAL;

s.get(1,y);

assert((x == ARRAY[0] && y == ARRAY[1]) ||

(x == ARRAY[1] && y == ARRAY[0]));

} break; case 23: {

s.insert(ARRAY[3]);

s.insert(ARRAY[1]);

s.insert(ARRAY[0]);

s.insert(ARRAY[4]);

s.insert(ARRAY[2]);

assert(s.size() == 5);

} break; case 24: {

s.insert(ARRAY[3]);

s.insert(ARRAY[1]);

s.insert(ARRAY[0]);

s.insert(ARRAY[4]);

s.insert(ARRAY[2]);

ItemType a[5] = { SOMEVAL, SOMEVAL, SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 5; k++)

s.get(k, a[k]);

assert(isPermutation(a, ARRAY, 5));

} break; case 25: {

s.insert(ARRAY[1]);

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[1]);

s.insert(ARRAY[0]);

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

} break; case 26: {

s.insert(ARRAY[1]);

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[1]);

s.insert(ARRAY[0]);

ItemType a[2] = { SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 2; k++)

s.get(k, a[k]);

assert(isPermutation(a, ARRAY, 2));

} break; case 27: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.insert(ARRAY[3]);

s.insert(ARRAY[4]);

assert(s.erase(ARRAY[3]));

} break; case 28: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.insert(ARRAY[3]);

s.insert(ARRAY[4]);

s.erase(ARRAY[3]);

assert(s.size() == 4);

} break; case 29: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.insert(ARRAY[3]);

s.insert(ARRAY[4]);

assert(s.erase(ARRAY[0]) && s.size() == 4);

} break; case 30: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.insert(ARRAY[3]);

s.insert(ARRAY[4]);

assert(s.erase(ARRAY[4]) && s.size() == 4);

} break; case 31: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[0]);

s.insert(ARRAY[2]);

s.insert(ARRAY[1]);

assert(s.erase(ARRAY[1]) && s.size() == 2);

} break; case 32: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.insert(ARRAY[3]);

s.insert(ARRAY[1]);

s.erase(ARRAY[3]);

ItemType a[3] = { SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 3; k++)

s.get(k, a[k]);

assert(s.size() == 3 && isPermutation(a, ARRAY, 3));

} break; case 33: {

s.insert(ARRAY[0]);

s.insert(ARRAY[3]);

s.insert(ARRAY[2]);

s.insert(ARRAY[1]);

s.insert(ARRAY[3]);

s.erase(ARRAY[3]);

ItemType a[3] = { SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 3; k++)

s.get(k, a[k]);

assert(s.size() == 3 && isPermutation(a, ARRAY, 3));

} break; case 34: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.insert(ARRAY[3]);

assert(!s.erase(ARRAY[4]));

} break; case 35: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.insert(ARRAY[3]);

s.erase(ARRAY[4]);

assert(s.size() == 4);

} break; case 36: {

s.insert(SOMEVAL);

s.erase(SOMEVAL);

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

} break; case 37: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

ItemType x;

assert(!s.get(-1, x));

} break; case 38: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

ItemType x = SOMEVAL;

s.get(-1, x);

assert(x == SOMEVAL);

} break; case 39: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

ItemType x;

assert(! s.get(3, x));

} break; case 40: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

ItemType x = SOMEVAL;

s.get(3, x);

assert(x == SOMEVAL);

} break; case 41: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.erase(ARRAY[1]);

assert(!s.contains(DEFAULT));

} break; case 42: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.erase(ARRAY[1]);

assert(s.contains(ARRAY[2]));

} break; case 43: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.erase(ARRAY[2]);

assert(s.size() == 2 && !s.contains(ARRAY[2]));

} break; case 44: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

s.erase(ARRAY[1]);

s.insert(ARRAY[2]);

s.insert(ARRAY[3]);

assert(s.size() == 3 && s.contains(ARRAY[2]) &&

s.contains(ARRAY[3]));

} break; case 45: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

{

Set s2;

s2.insert(ARRAY[3]);

s2.insert(ARRAY[4]);

s.swap(s2);

assert(s.size() == 2 && s2.size() == 3);

}

} break; case 46: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

{

Set s2;

s2.insert(ARRAY[3]);

s2.insert(ARRAY[4]);

s.swap(s2);

ItemType x;

assert(s.contains(ARRAY[3]) && s.contains(ARRAY[4]));

for (size\_t k = 0; k < 3; k++)

assert(!s.contains(ARRAY[k]));

}

} break; case 47: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

{

Set s2;

s2.insert(ARRAY[3]);

s2.insert(ARRAY[4]);

int n = nasstcopys();

s.swap(s2);

int n2 = nasstcopys();

assert(n == n2);

}

} break; case 48: {

{

Set s2;

s2.insert(ARRAY[0]);

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

s2.erase(ARRAY[1]);

s2.insert(ARRAY[3]);

s2.erase(ARRAY[2]);

s2.insert(ARRAY[1]);

s2.erase(ARRAY[0]);

}

assert(true); // no corruption so bad that destruction failed

} break; case 49: {

{

Set s2;

s2.insert(ARRAY[0]);

s2.insert(ARRAY[1]);

Set s3(s2);

s3.insert(ARRAY[2]);

s3.erase(ARRAY[1]);

s3.insert(ARRAY[3]);

s3.erase(ARRAY[2]);

s3.insert(ARRAY[1]);

s3.erase(ARRAY[0]);

}

assert(true); // no corruption so bad that destruction failed

} break; case 50: {

int before = itemcount();

{

Set s2;

s2.insert(SOMEVAL);

assert(itemcount() > before);

}

assert(itemcount() == before);

} break; case 51: {

int before = itemcount();

{

Set s2;

s2.insert(ARRAY[0]);

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

int n2 = itemcount() - before;

Set s3(s2);

int n3 = itemcount() - before;

assert(n3 = 2\*n2);

}

} break; case 52: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

{

Set s2(s);

s2.insert(ARRAY[3]);

assert(s2.size() == s.size()+1);

}

} break; case 53: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

{

Set s2(s);

s2.insert(ARRAY[3]);

ItemType a[4] = { SOMEVAL, SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 4; k++)

s2.get(k, a[k]);

assert(s2.size() == 4 && isPermutation(a, ARRAY, 4));

}

} break; case 54: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

s.insert(ARRAY[2]);

{

Set s2(s);

s2.insert(ARRAY[3]);

ItemType a[3] = { SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 3; k++)

s.get(k, a[k]);

assert(s.size() == 3 && isPermutation(a, ARRAY, 3));

}

} break; case 55: {

{

Set s2;

s2.insert(ARRAY[0]);

s2.insert(ARRAY[1]);

Set s3;

s3.insert(ARRAY[3]);

s3.insert(ARRAY[4]);

s3 = s2;

s3.insert(ARRAY[2]);

s3.erase(ARRAY[1]);

s3.insert(ARRAY[3]);

s3.erase(ARRAY[2]);

s3.insert(ARRAY[1]);

s3.erase(ARRAY[0]);

}

assert(true); // no corruption so bad that destruction failed

} break; case 56: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

int before = itemcount();

{

Set s2;

s2.insert(ARRAY[2]);

s2.insert(ARRAY[3]);

s2.insert(ARRAY[4]);

int mid = itemcount();

s2 = s;

assert(itemcount() <= mid);

}

assert(itemcount() == before);

} break; case 57: {

int before = itemcount();

{

Set s2;

s2.insert(ARRAY[0]);

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

int n2 = itemcount() - before;

Set s3;

s3.insert(ARRAY[3]);

s3.insert(ARRAY[4]);

s3 = s2;

int n3 = itemcount() - before;

assert(n3 = 2\*n2);

}

} break; case 58: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

{

Set s2;

s2.insert(ARRAY[2]);

s2.insert(ARRAY[3]);

s2.insert(ARRAY[4]);

s2 = s;

s2.insert(ARRAY[5]);

assert(s2.size() == s.size()+1);

}

} break; case 59: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

{

Set s2;

s2.insert(ARRAY[3]);

s2.insert(ARRAY[4]);

s2.insert(ARRAY[5]);

s2 = s;

s2.insert(ARRAY[2]);

ItemType a[3] = { SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 3; k++)

s2.get(k, a[k]);

assert(s2.size() == 3 && isPermutation(a, ARRAY, 3));

}

} break; case 60: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

{

Set s2;

s2.insert(ARRAY[3]);

s2.insert(ARRAY[4]);

s2.insert(ARRAY[5]);

s2 = s;

s2.insert(ARRAY[2]);

ItemType a[2] = { SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 2; k++)

s.get(k, a[k]);

assert(s.size() == 2 && isPermutation(a, ARRAY, 2));

}

} break; case 61: {

{

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[3]);

s2.insert(ARRAY[2]);

int n2 = itemcount();

s2 = s2;

assert(itemcount() == n2);

assert(s2.size() == 3 && s2.contains(ARRAY[1]) &&

s2.contains(ARRAY[2]) && s2.contains(ARRAY[3]));

}

assert(true); // no corruption so bad that destruction failed

} break; case 62: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

Set s3;

unite(s,s2,s3);

assert(s3.size() == s.size());

ItemType a[2] = { SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 2; k++)

s3.get(k, a[k]);

assert(isPermutation(a, ARRAY, 2));

} break; case 63: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

Set s3;

unite(s2,s,s3);

assert(s3.size() == s.size());

ItemType a[2] = { SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 2; k++)

s3.get(k, a[k]);

assert(isPermutation(a, ARRAY, 2));

} break; case 64: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

Set s3;

unite(s,s2,s3);

assert(s3.size() == 3);

ItemType a[3] = { SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 3; k++)

s3.get(k, a[k]);

assert(isPermutation(a, ARRAY, 3));

} break; case 65: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

Set s3;

s3.insert(ARRAY[3]);

unite(s,s2,s3);

assert(s3.size() == 3);

ItemType a[3] = { SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 3; k++)

s3.get(k, a[k]);

assert(isPermutation(a, ARRAY, 3));

} break; case 66: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

unite(s,s2,s);

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

ItemType a[3] = { SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 3; k++)

s.get(k, a[k]);

assert(isPermutation(a, ARRAY, 3));

} break; case 67: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

unite(s,s2,s2);

assert(s2.size() == 3);

ItemType a[3] = { SOMEVAL, SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 3; k++)

s2.get(k, a[k]);

assert(isPermutation(a, ARRAY, 3));

} break; case 68: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

Set s3;

subtract(s,s2,s3);

assert(s3.size() == s.size());

ItemType a[2] = { SOMEVAL, SOMEVAL };

for (size\_t k = 0; k < 2; k++)

s3.get(k, a[k]);

assert(isPermutation(a, ARRAY, 2));

} break; case 69: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

Set s3;

subtract(s2,s,s3);

assert(s3.size() == 0);

} break; case 70: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

Set s3;

subtract(s,s2,s3);

assert(s3.size() == 1);

ItemType x;

s3.get(0, x);

assert(x == ARRAY[0]);

} break; case 71: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

Set s3;

s3.insert(ARRAY[3]);

s3.insert(ARRAY[4]);

subtract(s,s2,s3);

assert(s3.size() == 1);

ItemType x;

s3.get(0, x);

assert(x == ARRAY[0]);

} break; case 72: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

subtract(s,s2,s);

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

ItemType x;

s.get(0, x);

assert(x == ARRAY[0]);

} break; case 73: {

s.insert(ARRAY[0]);

s.insert(ARRAY[1]);

Set s2;

s2.insert(ARRAY[1]);

s2.insert(ARRAY[2]);

subtract(s,s2,s2);

assert(s2.size() == 1);

ItemType x;

s2.get(0, x);

assert(x == ARRAY[0]);

} break; case 74: {

const int NITEMS = 2000;

for (size\_t k = 0; k < NITEMS; k++)

assert(s.insert(ItemType(k)));

assert(s.size() == NITEMS);

}

}

}

int main()

{

cout << "Enter test number: ";

int n;

cin >> n;

testone(n);

cout << "Passed" << endl;

}