**Homework 4**

**Time due: 9:00 PM Tuesday, May 29**

1. The files [Set.h](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/Set.h.txt) and [Set.cpp](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/Set.cpp.txt) contain the definition and implementation of Set implemented using a doubly-linked list. A client who wants to use a Set has to change the typedef in Set.h, and within one source file, cannot have two Sets containing different types.

Change Set to be a class template, so that a client can say

#include "Set.h"

#include <string>

using std::string;

...

Set<int> si;

Set<string> ss;

si.insert(5);

ss.insert("Maroon 5");

...

Also, change unite and subtract to be function templates.

(Hint: Transforming the typedef-based solution is a mechanical task that takes five minutes if you know what needs to be done. What makes this problem non-trivial for you is that you haven't done it before; the syntax for declaring templates is new to you, so you may not get it right the first time. Have you looked at Chapter 8, pp. 419-426?)

(Hint: The template typename parameter doesn't have to be named T; it can be a name of your choosing. You might find that by choosing ItemType you'll have fewer changes to make.)

(Hint: The Node class nested in the Set class can talk about the template parameters of the Set class; it should not itself be a template class.)

Because of most current compilers' limitations, the definition *and* implementation of your Set class template should be in just one file, Set.h, which is all that you will turn in for this problem. Although the implementation of a non-template non-inline function should not be placed in a header file (because of linker problems if that header file were included in multiple source files), the implementation of a template function, whether or not it's declared inline, *can* be in a header file.

There's a C++ language technicality that your compiler may or may not enforce. It relates to a type declared inside a class template, like N below:

template <typename T>

class S

{

...

struct N

{

...

};

N\* f();

...

};

If we attempt to implement f this way:

template <typename T>

S<T>::N\* S<T>::f() // shouldn't compile

{

...

}

the technicality requires the compiler to not recognize S<T>::N as a type name; it must be announced as a type name this way:

template <typename T>

typename S<T>::N\* S<T>::f() // OK

{

...

}

1. Consider this program:
2. #include "Set.h" // class template from problem 1
3. #include <string>
4. using namespace std;
5. class Bottle
6. {
7. public:
8. Bottle(string c) : m\_contents(c) {}
9. Bottle() : m\_contents("Empty") {}
10. string contents() const { return m\_contents; }
11. private:
12. string m\_contents;
13. };
14. int main()
15. {
16. Set<int> si;
17. si.insert(7); // OK
18. Set<Bottle> sb;
19. sb.insert(Bottle("7-Up")); // error!
20. }

Explain in a sentence or two why the call to Set<Bottle>::insert causes at least one compilation error. (Notice that the call to Set<int>::insert is fine.) Don't just transcribe a compiler error message; your answer must indicate you understand the reason.

1. A file has a name. A file is either a plain file (like a text file, an image file, a C++ source program, etc.) or a directory. Directories contain zero or more files. The following program reflects this structure:
2. #include <iostream>
3. #include <string>
4. #include <vector>
5. using namespace std;
6. class File
7. {
8. public:
9. File(string nm) : m\_name(nm) {}
10. virtual ~File() {};
11. string name() const { return m\_name; }
12. virtual bool add(File\* f) = 0;
13. virtual const vector<File\*>\* files() const = 0;
14. private:
15. string m\_name;
16. };
17. class PlainFile : public File
18. {
19. public:
20. PlainFile(string nm) : File(nm) {}
21. virtual bool add(File\* f) { return false; }
22. virtual const vector<File\*>\* files() const { return NULL; }
23. };
24. class Directory : public File
25. {
26. public:
27. Directory(string nm) : File(nm) {}
28. virtual ~Directory();
29. virtual bool add(File\* f) { m\_files.push\_back(f); return true; }
30. virtual const vector<File\*>\* files() const { return &m\_files; }
31. private:
32. vector<File\*> m\_files;
33. };
34. Directory::~Directory()
35. {
36. for (int k = 0; k < m\_files.size(); k++)
37. delete m\_files[k];
38. }
39. void listAllAuxiliary(string path, const File\* f)
40. {
41. *You will write this code.*
42. }
43. void listAll(const File\* f)
44. {
45. listAllAuxiliary("", f);
46. }
47. int main()
48. {
49. Directory\* d1 = new Directory("Fun");
50. d1->add(new PlainFile("party.jpg"));
51. d1->add(new PlainFile("beach.jpg"));
52. d1->add(new PlainFile("skitrip.jpg"));
53. Directory\* d2 = new Directory("Work");
54. d2->add(new PlainFile("seaslab.jpg"));
55. Directory\* d3 = new Directory("My Pictures");
56. d3->add(d1);
57. d3->add(new PlainFile("me.jpg"));
58. d3->add(new Directory("Miscellaneous"));
59. d3->add(d2);
60. listAll(d3);
61. delete d3;
62. }

When the listAll function is called from the main routine above, the following output should be produced (the first line written is /My Pictures, not an empty line):

/My Pictures

/My Pictures/Fun

/My Pictures/Fun/party.jpg

/My Pictures/Fun/beach.jpg

/My Pictures/Fun/skitrip.jpg

/My Pictures/me.jpg

/My Pictures/Miscellaneous

/My Pictures/Work

/My Pictures/Work/seaslab.jpg

This is a list, one per line, of the complete pathname for listAll's argument and, if the argument is a directory, everything in that directory. A pathname starts with / and has / separating pathname components.

* 1. You are to write the code for listAllAuxiliary to make this happen. You must not use any additional container (such as a stack), and listAllAuxiliary must be recursive. You must not use any global variables or variables declared with the keyword static, and you must not modify any of the code we have already written or add new functions. You may use a loop to traverse the vector; you must not use loops to avoid recursion.

Here's a useful function to know: the standard library string class has a + operator that concatenates strings and/or characters. For example,

string s("Hello");

string t("there");

string u = s + ", " + t + '!';

// Now u has the value "Hello, there!"

It's also useful to know that if you choose to traverse an STL container using some kind of iterator, then if the container is const, you must use a const\_iterator:

void f(const list<int>& c) // c is const

{

for (list<int>::const\_iterator it = c.begin(); it != c.end(); it++)

cout << \*it << endl;

}

(Of course, a vector can be traversed either by using some kind of iterator, or by using operator[] with an integer argument).

For this problem, you will turn a file named list.cpp with the body of the listAllAuxiliary function, from its "void" to its "}", no more and no less. Your function must compile and work correctly when substituted into the program above. (Make sure you spell and capitalize "listAllAuxiliary" correctly.)

* 1. We introduced the function listAllAuxiliary. Why could you not solve this problem given the constraints in part a if you had to implement listAll (with its one parameter) as the recursive function?
  2. Suppose we have a list of N world cities, numbered from 0 to N-1. The two-dimensional array of doubles dist holds the airline distance between each pair of cities: dist[i][j] is the distance between city i and city j.

Now, for every pair of cities i and j, we'd like to consider all the flights between the two that make one stop in a third city k, and record which city k yields the shortest distance traveled in a one-stop flight between city i and city j that passes through city k. Here's the code:

const int N = *some value*;

assert(N > 2); // algorithm fails if N <= 2

double dist[N][N];

...

int bestMidPoint[N][N];

for (int i = 0; i < N; i++)

{

bestMidPoint[i][i] = -1; // one-stop trip to self is silly

for (int j = 0; j < N; j++)

{

if (i == j)

continue;

int minDist = *maximum possible integer*;

for (int k = 0; k < N; k++)

{

if (k == i || k == j)

continue;

int d = dist[i][k] + dist[k][j];

if (d < minDist)

{

minDist = d;

bestMidPoint[i][j] = k;

}

}

}

}

What is the time complexity of this algorithm, in terms of the number of basic operations (e.g., additions, assignments, comparisons) performed: Is it O(N), O(N log N), or what? Why? (Note: In this homework, whenever we ask for the time complexity, we care only about the high order term, so don't give us answers like O(N2+4N).)

* 1. The algorithm in part a doesn't take advantage of the symmetry of distances: for every pair of cities i and j, dist[i][j] == dist[j][i]. We can skip a lot of operations and compute the best midpoints more quickly with this algorithm:
  2. const int N = *some value*;
  3. assert(N > 2); // algorithm fails if N <= 2
  4. double dist[N][N];
  5. ...
  6. int bestMidPoint[N][N];
  7. for (int i = 0; i < N; i++)
  8. {
  9. bestMidPoint[i][i] = -1; // one-stop trip to self is silly
  10. for (int j = 0; j < **i**; j++) **// loop limit is now i, not N**
  11. {
  12. int minDist = *maximum possible integer*;
  13. for (int k = 0; k < N; k++)
  14. {
  15. if (k == i || k == j)
  16. continue;
  17. int d = dist[i][k] + dist[k][j];
  18. if (d < minDist)
  19. {
  20. minDist = d;
  21. bestMidPoint[i][j] = k;
  22. **bestMidPoint[j][i] = k;**
  23. }
  24. }
  25. }
  26. }

What is the time complexity of this algorithm? Why?

* 1. Here again is the (non-template) non-member unite function for Sets from [Set.cpp](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/Set.cpp.txt):
  2. void unite(const Set& s1, const Set& s2, Set& result)
  3. {
  4. const Set\* sp = &s2;
  5. if (&result == &s1)
  6. {
  7. if (&result == &s2)
  8. return;
  9. }
  10. else if (&result == &s2)
  11. sp = &s1;
  12. else
  13. {
  14. result = s1;
  15. if (&s1 == &s2)
  16. return;
  17. }
  18. for (int k = 0; k < sp->size(); k++)
  19. {
  20. ItemType v;
  21. sp->get(k, v);
  22. result.insert(v);
  23. }
  24. }

Assume that s1, s2, and the old value of result each have N elements. In terms of the number of linked list nodes visited during the execution of this function, what is its worst case time complexity? Why?

* 1. Here is an implementation of a related member function. The call
  2. set3.unite(set1,set2);

sets set3 to the set union of set1 and set2. The implementation is

void Set::unite(const Set& s1, const Set& s2)

{

vector<ItemType> v;

// copy all items into v;

for (Node\* p1 = s1.m\_head->m\_next; p1 != s1.m\_head; p1 = p1->m\_next)

v.push\_back(p1->m\_data);

for (Node\* p2 = s2.m\_head->m\_next; p2 != s2.m\_head; p2 = p2->m\_next)

v.push\_back(p2->m\_data);

// sort v using an O(N log N) algorithm

sort(v.begin(), v.end());

// delete result nodes (other than the dummy node)

while (m\_head->next != m\_head)

doErase(m\_head->m\_next);

// copy unique items from v into result

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

{

if (k == 0 || v[k] != v[k-1]) // add non-duplicates

insertAtTail(v[k]);

}

// v is destroyed when function returns

}

Assume that s1, s2, and the old value of \*this each have N elements. In terms of the number of ItemType objects visited (either in linked list nodes or the vector) during the execution of this function, what is its time complexity? Why?

1. The file [sorts.cpp](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/sorts.cpp.txt) contains an almost complete program that creates a randomly ordered array, sorts it in a few ways, and reports on the elapsed times. Your job is to complete it and experiment with it.

You can run the program as is to get some results for the STL sort algorithm. The insertion sort result will be meaningless (and you'll get an assertion violation), because the insertion sort function right now doesn't do anything. That's one thing for you to write.

The objects in the array are not cheap to copy, which makes a sort that does a lot of moving objects around expensive. Your other task will be to create a vector of *pointers* to the objects, sort the pointers using the same criterion as was used to sort the objects, and then make one pass through the vector to put the objects in the proper order.

Your two tasks are thus:

* 1. Implement the insertion\_sort function.
  2. Implement the compareStorePtr function and the code in main to create and sort the array of pointers.

The places to make modifications are indicated by TODO: comments. You should not have to make modifications anywhere else. (Our solution doesn't.)

Try the program with about 10000 stores. Depending on the speed of your processor, this number may be too large or small to get meaningful timings in a reasonable amount of time. Experiment. Once you get insertion sort working, observe the O(N2) behavior by sorting, say, 10000, 20000, and 30000 stores.

To further observe the performance behavior of the STL sort algorithm, try sorting, say, 100000, 200000, and 300000 items, or even ten times as many. Since this would make the insertion sort tests take a long time, skip them by setting the TEST\_INSERTION\_SORT constant at the top of sorts.cpp to false.

Notice that if you run the program more than once, you may not get exactly the same timings each time. This is *not* because you're not getting the same sequence of random numbers each time; you are. Instead, factors like caching by the operating system are the cause.

**Turn it in**

By Monday, May 28, there will be a link on the class webpage that will enable you to turn in this homework. Turn in one zip file that contains your solutions to the homework problems. The zip file must contain four files:

* Set.h, a C++ header file with your definition and implementation of the Set class template for problem 1. This test program that we will try with your header must build and execute successfully:
* #include "Set.h"
* #include <iostream>
* #include <string>
* #include <cassert>
* using namespace std;
* void test()
* {
* Set<int> si;
* Set<string> ss;
* assert(si.empty());
* assert(ss.size() == 0);
* assert(si.insert(10));
* assert(ss.insert("hello"));
* assert(si.contains(10));
* assert(si.erase(10));
* string s;
* assert(ss.get(0, s) && s == "hello");
* Set<string> ss2(ss);
* ss.swap(ss2);
* ss = ss2;
* unite(si,si,si);
* unite(ss,ss2,ss);
* subtract(si,si,si);
* subtract(ss,ss2,ss);
* cout << "Passed all tests" << endl;
* }
* int main()
* {
* test();
* cout << "Passed all tests" << endl;
* }
* list.cpp, a C++ source file with implementation of the listAllAuxiliary function for problem 3a.
* sorts.cpp, a C++ source file with your solution to problem 6.
* hw.doc, hw.docx, or hw.txt, a Word document or a text file with your solutions to problems 2, 3b, 4a, 4b, 5a, and 5b.