Prianna Ahsan

UID: 704068040

03/02/2013

Homework #4

2. Sample code:

#include "Multiset.h" // class template from problem 1

#include <string>

#include <cassert>

using namespace std;

class URL

{

public:

URL(string i) : m\_id(i) {}

URL() : m\_id("http://cs.ucla.edu") {}

string id() const { return m\_id; }

private:

string m\_id;

};

int main()

{

Multiset<int> mi;

mi.insert(7); // OK

Multiset<string> ms;

ms.insert("http://www.symantec.com"); // OK

Multiset<URL> mu;

mu.insert(URL("http://www.symantec.com")); // error!

}

The last line of the above code will generate compilation errors because there are no comparison operators defined in class URL. As a result, the class Multiset cannot determine when two URLs are not equal to one another: e.g., URL1 != URL2 is undefined. Defining the != operator in class URL (e.g, if (URL1.m\_id != URL2.m\_id) return true; else return false;) would allow such a comparison to be made. We don’t have to define != to compare m\_id, since it’s is a string and string types in C++ have a defined != operator.

3b. Attempting to use listAll recursively would have resulted in only the called object’s m\_name variable being printed. You could append (or prepend) the called object’s m\_name var to the calling object’s m\_name var with a ‘/’ in between, but it still wouldn’t produce the desired output. Instead it would print something along the lines of

File/New

New/Window

File/Open

etc…

**4a. Sample code (with my comments in green, answer underlined):**

const int N = some value;

// The assert below runs once.

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

double dist[N][N];

...

int bestMidPoint[N][N];

for (int i = 0; i < N; i++) // This loop will run N times

{

// The statement below sets all diagonals to (-1).

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

for (int j = 0; j < N; j++) // This loop will run N times

{

if (i == j) // i == j N times (NxN matrix with N^2 entries).

continue;

int minDist = maximum possible integer;

for (int k = 0; k < N; k++) // This loop will run N^2-N times.

{ // The statement below is mutually exclusive. k will never

// equal both i & j, since the inner most loop only runs when

// i != j. This statement will execute N times.

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

continue;

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

if (d < minDist)

{

minDist = d;

bestMidPoint[i][j] = k;

…

All together, this algorithm will run in N3 time: N2\*N(N-1) is O(N3).

**4b. Sample code (with my comments in green, answer underlined):**

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++) // Runs N times.

{

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

// The loop below executes N-1 times (worst case).

for (int j = 0; j < i; j++) // loop limit is now i, not N

{

int minDist = maximum possible integer;

for (int k = 0; k < N; k++) // Still runs N times.

{

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

continue;

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

if (d < minDist)

{

minDist = d;

bestMidPoint[i][j] = k;

bestMidPoint[j][i] = k;

…

All together, this algorithm will run N\*N-1\*N times, in the worst case, which is O(N3). Not much of an improvement.

**5a. Sample codes (with my comments in green):**

void uniqueIntersect(const Multiset& m1, const Multiset& m2, Multiset& result)

{

Multiset res;

for (int k = 0; k != m1.uniqueSize(); k++) // Runs N times.

{

ItemType x;

// get() performs a binary search, so the operation below takes

// log2(N) time every time the loop runs.

m1.get(k, x);

// contains() calls find(), which performs a linear search.

if (m2.contains(x))

// insert() calls find(), which performs a linear search in // N time.

res.insert(x);

}

result.swap(res); // runs once.

}

Worst case: The outer loop runs N times, get() executes log2(N) times per iteration, contains() executes N times per iteration, and insert() (which only executes if contains() returns true) runs N times as well (worst case: ms1 == ms2). Swap() runs in constant time, and simply swaps head nodes. All together, this function runs in N\*log2(N)\*N\*N times, or O(N3log(N)).

**5b. Relevant sample code (my comments in green, answer underlined):**

// N times

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

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

// N times

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

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

sort(v.begin(), v.end()); // N log N

…

for (size\_t k = 1; k < v.size(); k++) // 2N times

{

if (v[k] == v[k-1]) // Worst case: Every other iteration.

{

Node\* toUpdate;

if (p != m\_head) // Worst case: Every other iteration.

{

}

else // Constant time

{…

}

…

}

}

// delete excess result nodes

if (p != m\_head) // 0 times if N identical items, N times if 0.

…

N+N+N\*log2(N)+2N+N+N… All together, O(N\*log2N) in the average case. A improvement of at least 2 orders of magnitude over the code in problem 5a.