Homework 4 Questions

2. Consider this program…

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

#include <string>

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 call to Multiset<URL>::insert causes compilation errors because the insert function calls the find function (defined earlier in the code I grabbed from the CS32 website) and the find function uses a != operator, attempting to compare two URLs. This causes an error because there is no defined != operator inside the URL class.

3b. We need the 2 parameter version because we need the second parameter string to keep track of the current path and pass that onto the next recursive call which will add on to the path.

4. Suppose we have a list of N world cities…dist [i][j] is the distance between city i and city j.

a. what is the big O of the given method? ----- O(N3)

for (int i = 0; i < N; i++)**//worst case N**

{

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

for (int j = 0; j < N; j++)**//worst case N**

{

if (i == j)**//ignore**

continue;

int minDist = maximum possible integer;

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

{

if (k == i || k == j)**//worst case this doesn’t happen**

continue;

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

if (d < minDist)

{

minDist = d;

bestMidPoint[i][j] = k;

}

}

}

}

b. What is time complexity of this algorithim? 🡪 It’s the same ------ O(N3), although it may decrease the number of operations, the highest valued exponent is still N cubed.

5 .Consider the Multiset class…

a. Consider this code

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

{

Multiset res;

for (int k = 0; k != m1.uniqueSize(); k++)**//uniqueSize() is an O(N)**

{ **//operation,for loop is O(N2)**

ItemType x;

m1.get(k, x);**//worst case, this is an O(N) operation**

if (m2.contains(x))**//worst case, another O(N)**

res.insert(x);**//worst case, O(N)**

}

result.swap(res);**//constant time**

}

The time complexity for this algorithm is O(N2) because each checking of the condition of the loop runs O(1) – (checking the unique size), and so does the loop itself, resulting in a for loop that has O(N) operations alone. The contents of the for loop’s highest order operation is O(N), so when we multiply, we are resulted with O(N2) for the entire algorithm.

b. consider this code…

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

{

vector<ItemType> v;

v.reserve(m1.uniqueSize() + m2.uniqueSize());**//constant time, ignore.**

// copy all items into v;**//O(N) operations, ignore…**

for (Node\* p1 = m1.m\_head->m\_next; p1 != m1.m\_head; p1 = p1->m\_next)

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

for (Node\* p2 = m2.m\_head->m\_next; p2 != m2.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());**//N LOG N**

// Items in the intersection will be those that appear twice in

// v, adjacent to each other.

// Copy one instance of those items from v into \*this.

m\_size = 0;

Node\* p = m\_head->next;

for (size\_t k = 1; k < v.size(); k++)**O(N) loop (“2N”)**

{

if (v[k] == v[k-1])

{

Node\* toUpdate;

if (p != m\_head)

{

toUpdate = p; // reuse existing node

p = p->m\_next;

}

else

{

// Insert new node at tail of result

toUpdate = new Node;

toUpdate->m\_next = m\_head;

toUpdate->m\_prev = m\_head->m\_prev;

toUpdate->m\_prev->m\_next = toUpdate;

toUpdate->m\_next->m\_prev = toUpdate;

}

toUpdate->m\_value = v[k];

toUpdate->m\_count = 1;

m\_size++;

}

}

// delete excess result nodes

if (p != m\_head)

{

m\_head->m\_prev = p->m\_prev;

p->m\_prev->m\_next = m\_head;

do

{

Node\* toBeDeleted = p;

p = p->m\_next;

delete toBeDeleted;

} while (p != m\_head);

}

m\_uniqueSize = m\_size;

// v is destroyed when function returns

}

The time complexity for this algorithm is O(NlogN). This is because the sort is the highest order operation in the entire algorithm, the rest of the code contained a bunch of O(N) algorithms. This is better than the algorithm in 5a because it saves many steps. The entire process can be quickened when working with a sorted list.