Eideen Mozaffari

UID: 105988436

Homework 4 Report

**Problem 1.e:** Explain in a sentence or two what happens during the execution of test case 3 that eventually leads to test case 3 failing.

Solution:

Once k is 2, the program pushes 5 more elements onto the vector. Presumably, since the vector has grown by a factor of 2, the compiler needs to allocate memory for a new vector with a higher capacity to store these 10 elements. However, this results in invalidation: the iterator that was being used on the original vector is now invalidated, since the vector has been moved in memory. Dereferencing the iterator when k is 3, after the vector has changed location in memory, results in undefined behavior, causing allValid to turn false and ultimately fail the test case.

**Problem 3:** Explain in a sentence or two why the call to the one-argument form of Sequence<Coord>::insert causes at least one compilation error.

Solution:

The one-argument form of insert will not compile because the compiler is not told how to compare two coordinates, since we have not overloaded the < operator, or even the == or > operators for that matter. The one-argument form of insert relies on sifting through elements in the sequence, until it finds an element less than or equal to the object passed as the parameter, at which point the object will be inserted (the two-argument form of insert does no comparisons, so there is no compile error there). Our Coordinate class has not defined how to compare Coordinates, and Coordinates are a non-primitive type, so the compiler will respond with a compile error.

**Problem 4.b:** We introduced the two-parameter overload of listAll. Why could you not solve this problem given the constraints in part a if we had only a one-parameter listAll, and you had to implement it as the recursive function?

Solution:

Since we were given the constraints in part a, specifically not being able to use any additional container and having to implement it as a recursive function, there is no way to have a variable that keeps track of the path to a submenu for each recursive call, using only a pointer to a MenuItem as the only parameter for the function. Having two parameters allows us to pass the path so far as a parameter and then append a /submenu to it. With one parameter, we cannot keep track of the path that has been traversed up until when we want to append another /submenu.

**Problem 5.a:** 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(N3+4N2).)

Solution:

The time complexity is O(N^3). The most inner loop runs O(N) times, the second most inner loop runs O(N^2) times, and the outer loop runs O(N^3) times. Individually, all three loops have time complexity O(N) because the stopping condition is once the iterator reaches N items. All other operations have constant O(1) time complexity, since accessing an element in an array takes constant time, and the Boolean checking takes constant time. By multiplication principle, we get

O(N) \* O(N) \* O(N) = O(N^3) as the overall time complexity.

**Problem 5.b:** What is the time complexity of this algorithm? Why?

Solution:

The time complexity is also O(N^3). The most inner loop runs O(N) times, the second most inner loop runs O(i)\*O(N) times, and the outer loop runs O(i)\*O(N^2) times. The difference in this algorithm is that the second loop has stopping condition of once the iterator reaches i. Although for each iteration i is a constant, i is still dependent on N. Summing up the number of times the second loop will run, we get (N-1)\*(N) / 2 by arithmetic sum formula, which is ≈ O(N^2), ignoring coefficients and lower order terms. All other operations have constant O(1) time complexity in this scenario, since accessing an element in an array takes constant time, and the Boolean checking takes constant time. By multiplication principle, we get

O(N) \* O(i) \* O(N) = O((N-1)\*N / 2) \* O(N) ≈ O(N^2) \* O(N) = O(N^3) as the overall time complexity.

**Problem 6.a:** Assume that seq1, seq2, 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 time complexity? Why?

Solution:

void concatReverse(const Sequence& seq1, const Sequence& seq2, Sequence& result)

{

Sequence res;

for (int k = seq1.size() - 1; k >= 0; k--) 🡺 O(N^2)

{

ItemType v;

seq1.get(k, v); 🡺 O(k) ≈ O(N)

res.insert(res.size(), v); 🡺 O(N)

}

for (int k = seq2.size() - 1; k >= 0; k--) 🡺 O(N^2)

{

ItemType v;

seq2.get(k, v); 🡺 O(k) ≈ O(N)

res.insert(res.size(), v); 🡺 O(N)

}

result.swap(res); 🡺 O(1)

}

The time complexity is O(N^2). Inside each loop, there are O(N) + O(N) = O(2N) ≈ O(N) operations. Since the loop runs N times, by multiplication principle, each loop has time complexity O(N) \* O(N) = O(N^2). The swap function takes constant time O(1) because we are only swapping pointers. Since there are two loops, by addition principle, we get

O(N^2) + O(N^2) = O(2 N^2) ≈ O(N^2) as the overall time complexity.

**Problem 6.b:** Assume that seq1, seq2, and the old value of \*this each have about N elements. In terms of the number of linked list nodes visited during the execution of this function, what is its time complexity? Why? Is it the same, better, or worse, than the implementation in part a?

Solution:

void Sequence::concatReverse(const Sequence& seq1, const Sequence& seq2)

{

Sequence res;

for (Node\* p = seq1.m\_head->m\_prev; p != seq1.m\_head; p = p->m\_prev)🡺O(N)

res.insertBefore(res.m\_head, p->m\_value); 🡺 O(1)

for (Node\* p = seq2.m\_head->m\_prev; p != seq2.m\_head; p = p->m\_prev)🡺O(N)

res.insertBefore(res.m\_head, p->m\_value); 🡺 O(1)

// Swap \*this with res

swap(res); 🡺 O(1)

// Old value of \*this (now in res) is destroyed when function returns.

}

The time complexity is O(N). Each insertion takes linear time O(1) because insertBefore takes in a pointer to the address of the insertion, allowing us to skip the process of iterating through our linked list to reach the correction position and rather already arrive at the correct address. Thus, insertBefore is a constant time function. So, since each loop runs N times, by multiplication principle, each loop has time complexity O(N). Since there are two loops, by addition principle, we get

O(N) + O(N) = O(2N) ≈ O(N) as the overall time complexity.

This implementation of concatReverse is better than the one in part a) because it has a lower time complexity.