Winter 2017 CS 32

**Homework 4 Solution**

|  |  |
| --- | --- |
| [Problem 1](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/4/solution.html#P1) | [Problem 4](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/4/solution.html#P4) |
| [Problem 2](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/4/solution.html#P2) | [Problem 5](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/4/solution.html#P5) |
| [Problem 3](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/4/solution.html#P3) | [Problem 6](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/4/solution.html#P6) |

**Problem 1:**

// Sequence.h

#ifndef SEQUENCE\_INCLUDED

#define SEQUENCE\_INCLUDED

**template <typename ItemType>**

class Sequence

{

public:

Sequence(); // Create an empty sequence (i.e., one whose size() is 0).

bool empty() const; // Return true if the sequence is empty, otherwise false.

int size() const; // Return the number of items in the sequence.

bool insert(int pos, const ItemType& value);

// Insert value into the sequence so that it becomes the item at

// position pos. The original item at position pos and those that

// follow it end up at positions one higher than they were at before.

// Return true if 0 <= pos <= size() and the value could be

// inserted. (It might not be, if the sequence has a fixed capacity

// (e.g., because it's implemented using a fixed-size array) and is

// full.) Otherwise, leave the sequence unchanged and return false.

// Notice that if pos is equal to size(), the value is inserted at the

// end.

int insert(const ItemType& value);

// Let p be the smallest integer such that value <= the item at

// position p in the sequence; if no such item exists (i.e.,

// value > all items in the sequence), let p be size(). Insert

// value into the sequence so that it becomes the item at position

// p. The original item at position p and those that follow it end

// up at positions one higher than before. Return p if the value

// was actually inserted. Return -1 if the value was not inserted

// (perhaps because the sequence has a fixed capacity and is full).

bool erase(int pos);

// If 0 <= pos < size(), remove the item at position pos from

// the sequence (so that all items that followed this item end up at

// positions one lower than they were at before), and return true.

// Otherwise, leave the sequence unchanged and return false.

int remove(const ItemType& value);

// Erase all items from the sequence that == value. Return the

// number of items removed (which will be 0 if no item == value).

bool get(int pos, ItemType& value) const;

// If 0 <= pos < size(), copy into value the item at position pos

// in the sequence and return true. Otherwise, leave value unchanged

// and return false.

bool set(int pos, const ItemType& value);

// If 0 <= pos < size(), replace the item at position pos in the

// sequence with value and return true. Otherwise, leave the sequence

// unchanged and return false.

int find(const ItemType& value) const;

// Let p be the smallest integer such that value == the item at

// position p in the sequence; if no such item exists, let p be -1.

// Return p.

void swap(Sequence**<ItemType>**& other);

// Exchange the contents of this sequence with the other one.

// Housekeeping functions

~Sequence();

Sequence(const Sequence**<ItemType>**& other);

Sequence**<ItemType>**& operator=(const Sequence**<ItemType>**& rhs);

private:

// Representation:

// a circular doubly-linked list with a dummy node.

// m\_head points to the dummy node.

// m\_head->m\_prev->m\_next == m\_head and m\_head->m\_next->m\_prev == m\_head

// m\_size == 0 iff m\_head->m\_next == m\_head->m\_prev == m\_head

// if m\_size > 0

// m\_head->next points to the node at position 0.

// m\_head->prev points to the node at position m\_size-1.

struct Node

{

ItemType m\_value;

Node\* m\_next;

Node\* m\_prev;

};

Node\* m\_head;

int m\_size;

void createEmpty();

// Create an empty list. (Should be called only by constructors.)

void insertBefore(Node\* p, const ItemType& value);

// Insert value in a new Node before Node p, incrementing m\_size.

Node\* doErase(Node\* p);

// Remove the Node p, decrementing m\_size. Return the Node that

// followed p.

Node\* nodeAtPos(int pos) const;

// Return pointer to Node at position pos. If pos == m\_size, return

// m\_head. (Will be called only when 0 <= pos <= size().)

};

// Declarations of non-member functions

**template <typename ItemType>**

int subsequence(const Sequence**<ItemType>**& seq1, const Sequence**<ItemType>**& seq2);

// If seq2 is a contiguous subsequence of seq1, return the position in

// seq1 where that subsequence starts (the earliest such position if more

// than one). If not, or if seq2 is empty, return -1.

**template <typename ItemType>**

void interleave(const Sequence**<ItemType>**& seq1, const Sequence**<ItemType>**& seq2,

Sequence**<ItemType>**& result);

// Set result to a Sequence that interleaves seq1 and seq2.

// Inline implementations

**template <typename ItemType>**

inline

int Sequence**<ItemType>**::size() const

{

return m\_size;

}

**template <typename ItemType>**

inline

bool Sequence**<ItemType>**::empty() const

{

return size() == 0;

}

// Non-inline implementations

**template <typename ItemType>**

Sequence**<ItemType>**::Sequence()

{

createEmpty();

}

**template <typename ItemType>**

Sequence**<ItemType>**::~Sequence()

{

// Delete all Nodes from first non-dummy up to but not including

// the dummy

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

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

// Delete the dummy

delete m\_head;

}

**template <typename ItemType>**

Sequence**<ItemType>**::Sequence(const Sequence**<ItemType>**& other)

{

createEmpty();

// Copy all non-dummy other Nodes. (This sets m\_size.)

// Inserting each new node before the dummy node that m\_head points to

// puts the new node at the end of the list.

for (Node\* p = other.m\_head->m\_next; p != other.m\_head; p = p->m\_next)

insertBefore(m\_head, p->m\_value);

}

**template <typename ItemType>**

Sequence**<ItemType>**& Sequence**<ItemType>**::operator=(const Sequence**<ItemType>**& rhs)

{

if (this != &rhs)

{

Sequence**<ItemType>** temp(rhs);

swap(temp);

}

return \*this;

}

**template <typename ItemType>**

bool Sequence**<ItemType>**::insert(int pos, const ItemType& value)

{

if (pos < 0 || pos > m\_size)

return false;

Node\* p = nodeAtPos(pos);

insertBefore(p, value);

return true;

}

**template <typename ItemType>**

int Sequence**<ItemType>**::insert(const ItemType& value)

{

// Find the Node before which to insert

Node\* p;

int pos;

for (p = m\_head->m\_next, pos = 0; p != m\_head &&

value > p->m\_value; p = p->m\_next, pos++)

;

// Insert the value there

insertBefore(p, value);

return pos;

}

**template <typename ItemType>**

bool Sequence**<ItemType>**::erase(int pos)

{

if (pos < 0 || pos >= m\_size)

return false;

Node\* p = nodeAtPos(pos);

doErase(p);

return true;

}

**template <typename ItemType>**

int Sequence**<ItemType>**::remove(const ItemType& value)

{

int count = 0;

// Walk through the list, erasing matching items

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

while (p != m\_head)

{

if (p->m\_value == value)

{

count++;

p = doErase(p); // p now points to successor of erased Node

}

else

p = p->m\_next; // p now points to successor of non-erased Node

}

return count;

}

**template <typename ItemType>**

bool Sequence**<ItemType>**::get(int pos, ItemType& value) const

{

if (pos < 0 || pos >= m\_size)

return false;

Node\* p = nodeAtPos(pos);

value = p->m\_value;

return true;

}

**template <typename ItemType>**

bool Sequence**<ItemType>**::set(int pos, const ItemType& value)

{

if (pos < 0 || pos >= m\_size)

return false;

Node\* p = nodeAtPos(pos);

p->m\_value = value;

return true;

}

**template <typename ItemType>**

int Sequence**<ItemType>**::find(const ItemType& value) const

{

// Walk through the list, keeping track of position

int pos = 0;

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

for ( ; p != m\_head && p->m\_value != value; p = p->m\_next, pos++)

;

return p == m\_head ? -1 : pos;

}

**template <typename ItemType>**

void Sequence**<ItemType>**::swap(Sequence**<ItemType>**& other)

{

// Swap head pointers

Node\* p = other.m\_head;

other.m\_head = m\_head;

m\_head = p;

// Swap sizes

int s = other.m\_size;

other.m\_size = m\_size;

m\_size = s;

}

**template <typename ItemType>**

void Sequence**<ItemType>**::createEmpty()

{

m\_size = 0;

// Create dummy node

m\_head = new Node;

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

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

}

**template <typename ItemType>**

void Sequence**<ItemType>**::insertBefore(Node\* p, const ItemType& value)

{

// Create a new node

Node\* newp = new Node;

newp->m\_value = value;

// Insert new item before p

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

newp->m\_next = p;

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

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

m\_size++;

}

**template <typename ItemType>**

**typename** Sequence**<ItemType>**::Node\* Sequence**<ItemType>**::doErase(Node\* p)

{

// Save pointer to p's successor

Node\* pnext = p->m\_next;

// Unlink p from the list and destroy it

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

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

delete p;

m\_size--;

return pnext;

}

**template <typename ItemType>**

**typename** Sequence**<ItemType>**::Node\* Sequence**<ItemType>**::nodeAtPos(int pos) const

{

Node\* p;

// If pos is closer to the head of the list, go forward to find it.

// Otherwise, start from tail and go backward.

if (pos <= m\_size / 2) // closer to head

{

p = m\_head->m\_next;

for (int k = 0; k != pos; k++)

p = p->m\_next;

}

else // closer to tail

{

p = m\_head;

for (int k = m\_size; k != pos; k--)

p = p->m\_prev;

}

return p;

}

**template <typename ItemType>**

int subsequence(const Sequence**<ItemType>**& seq1, const Sequence**<ItemType>**& seq2)

{

if (seq2.empty())

return -1;

// Walk through seq1

for (int pos = 0; pos <= seq1.size() - seq2.size(); pos++)

{

// Assume there's a match starting at pos

bool allMatched = true;

// Check if all corresponding positions match

for (int k = 0; k < seq2.size(); k++)

{

ItemType v1;

ItemType v2;

seq1.get(pos+k, v1);

seq2.get(k, v2);

if (v1 != v2)

{

allMatched = false;

break;

}

}

// If we never found a mismatch, we've found the match.

if (allMatched)

return pos;

}

// If we never found a match, there is none.

return -1;

}

**template <typename ItemType>**

void interleave(const Sequence**<ItemType>**& seq1, const Sequence**<ItemType>**& seq2, Sequence**<ItemType>**& result)

{

// Guard against the case that result is an alias for seq1 or seq2

// (i.e., that result is a reference to the same sequence that seq1 or

// seq2 refers to) by building the answer in a local variable res. When

// done, swap res with result; the old value of result (now in res) will

// be destroyed when res is destroyed.

Sequence**<ItemType>** res;

// Interleave elements until one or both sequences runs out.

int n1 = seq1.size();

int n2 = seq2.size();

int nmin = (n1 < n2 ? n1 : n2);

int resultPos = 0;

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

{

ItemType v;

seq1.get(k, v);

res.insert(resultPos, v);

resultPos++;

seq2.get(k, v);

res.insert(resultPos, v);

resultPos++;

}

// Append the remaining elements from the longer sequence. (If the

// sequences are the same length, this does nothing.)

const Sequence**<ItemType>**& s = (n1 > nmin ? seq1 : seq2);

int n = (n1 > nmin ? n1 : n2);

for (int k = nmin ; k < n; k++)

{

ItemType v;

s.get(k, v);

res.insert(resultPos, v);

resultPos++;

}

result.swap(res);

}

#endif // SEQUENCE\_INCLUDED

You could alternatively have implemented the member functions inside the class declaration:

template <typename ItemType>

class Sequence

{

...

int size() const

{

return m\_size;

}

...

};

Since for this problem you were already starting with a lot of code written with the implementations outside the class declaration, it was less work to leave their implementations outside.

**Problem 2:**

The instantiation of the one-argument form of Sequence<Complex>::insert contains the expression value > p->m\_value, where both operands are Complex. We never defined operator> for Complex operands (nor should we, since complex numbers don't have a natural ordering).

**Problem 3:**

1. void listAll(const MenuItem\* m, string path)
2. {
3. path += m->name();
4. if (!path.empty())
5. {
6. cout << path << endl;
7. path += '/';
8. }
9. if (m->menuItems() != nullptr)
10. {
11. const vector<MenuItem\*>& menuItems = \*m->menuItems();
12. for (size\_t k = 0; k != menuItems.size(); k++)
13. listAll(menuItems[k], path);
14. }
15. }
16. Other ways to write the for loop are
17. for (vector<MenuItem\*>::const\_iterator p = m->menuItems().begin();
18. p != m->menuItems().end(); p++)
19. listAll(\*p, path);

or (in C++11)

for (MenuItem\* subitem : m->menuItems()) // or for (auto subitem : m->menuItems())

listAll(path, subitem);

1. Without any static or global variables or any additional containers, there would be no way to keep track of the path from the root to each node of the tree.

**Problem 4:**

1. Consider the code in the k loop:
2. for (int k = 0; k < N; k++)
3. {
4. if (k == i || k == j)
5. continue;
6. if (isFriend[i][k] && isFriend[k][j])
7. numMutualFriends[i][j]++;
8. }

This involves one initialization (int k = 0), which we can ignore, since it's dominated by the N repetitions of everything else. The most work that each of the N iterations of the loop might do is a comparison (k < N), an increment (k++), two equality tests (k == i and k == j), an and test (&&), an increment (++), and 3 double subscriptings. These basic operations are each constant time, so this loop does no more than mN basic operations, for some constant m. This inner loop is O(N).

For the rest of this analysis, we won't be so meticulous: we'll drop low order terms where it won't affect the result.

The number of operations performed during one execution of the innermost loop body (which obviously accounts for most of the operations, since it's executed the most) is bounded by some constant m. In all, that body accounts for sum(i from 0 to N-1) of sum(j from 0 to N-1) of sum(k from 0 to N-1) of m operations.

sum(i from 0 to N-1) of {sum(j from 0 to N-1) of [sum(k from 0 to N-1) of m]} ~  
m \* sum(i from 0 to N-1) of {sum(j from 0 to N-1) of [sum(k from 0 to N-1) of 1]} ~  
m \* sum(i from 0 to N-1) of {sum(j from 0 to N-1) of N} ~  
m \* sum(i from 0 to N-1) of N\*N ~  
m \* N\*N\*N = O(N3)

1. Again, the innermost statement accounts for the most operations performed, and it accounts for sum(i from 0 to N-1) of sum(j from 0 to i-1) of sum(k from 0 to N-1) of m operations. We'll retain the constant of proportionality just to get a feel for how much faster this can be. (We assume m is about the same as Part a.)

sum(i from 0 to N-1) of {sum(j from 0 to i-1) of [sum(k from 0 to N-1) of m]} ~  
m \* sum(i from 0 to N-1) of {sum(j from 0 to i-1) of [sum(k from 0 to N-1) of 1]} ~  
m \* sum(i from 0 to N-1) of {sum(j from 0 to i-1) of N} ~  
m \* sum(i from 0 to N-1) of i\*N} ~  
m \* N \* sum(i from 0 to N-1) of i} ~  
m \* N \* ((N-1) \* N / 2)

Since ((N-1) \* N / 2) ~ N\*N/2, the full sum is about (m/2) \* N\*N\*N = O(N3). For large N, this algorithm is about twice as fast as the one in Part a, but it's still order N3; doubling the size of a problem increases the running time about eightfold.

**Problem 5:**

1. First, notice that our implementation of get and insert use the helper function nodeAtPos, which takes a length of time proportional to the distance between the desired position and the *nearest* end of the list. For position k in a list of size N, this is min(k, N-k).

Since our assumption was that each of seq1 and seq2 has N nodes, nmin will be N, and the second for loop will be executed zero times. (If one of the two sequences is a little longer than the other, then there will be fewer iterations of the first for loop, and more of the second; since the first does twice as much work per iteration as the second, this has the effect of reducing the running time. So the two sequences being of equal length is the worst case.) Since the size and swap member functions are constant time, it's clear the first for loop dominates the running time.

The statements in that loop are executed N times. Each time, each of the two calls to the get function visits min(k,N-k) nodes. Inserting at the end of the list visits one node as we've implemented it, so the time for each iteration is dominated by the two calls to get:

sum(k from 0 to N-1) of 2 \* min(k,N-k) ~  
2 \* sum(k from 0 to N-1) of min(k,N-k) ~

For k <= ½N, min(k,N-k) = k; for k >= ½N, min(k,N-k) = N-k. So,

2 \* sum(k from 0 to N-1) of min(k,N-k) ~  
2 \* {sum(k from 0 to ½N-1) of [k] + sum(k from ½N to N-1) of [N-k]} ~  
2 \* {(0+1+2+...+½N-1) + (½N+(½N-1)+...+1)} ~  
2 \* {N + 2 \* ½((½N)2 - ½N)} = O(N2)

1. Since swap is constant time, as are the other pointer manipulations outside the loops, the loops dominate the running time. The bodies of the loops call insertBefore; examining its code, we see it's constant time. Together, the loops iterate as many times as the length of the longer sequence; since each of those N iterations is O(1), the total running time of this version of interleave is O(N).

**Problem 6:**

Changes to the program as given are **bold**.

…

inline

bool compareStudentPtr(const Student\* lhs, const Student\* rhs)

{

return **compareStudent(\*lhs, \*rhs);**

}

…

void insertion\_sort(vector<Student>& s, bool comp(const Student&, const Student&))

{

**for (size\_t k = 1; k < s.size(); k++)**

**{**

**Student currentStudent(s[k]);**

**size\_t m = k;**

**for ( ; m > 0 && comp(currentStudent,s[m-1]); m--)**

**s[m] = s[m-1];**

**s[m] = currentStudent;**

**}**

}

…

// Create a auxiliary copy of students, to faciliate the later reordering.

// We create it in a local scope, so we also account for the destruction

// time.

{

vector<Student> auxStudents(students);

// Create a vector of Student pointers, and set each pointer

// to point to the corresponding Student in auxStudents.

**vector<Student\*> studentPtrs;**

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

**studentPtrs.push\_back(&auxStudents[k]);**

// Sort the vector of pointers using the STL sort algorithm

// with compareStudentPtr as the ordering relationship.

**sort(studentPtrs.begin(), studentPtrs.end(), compareStudentPtr);**

// Using the now-sorted vector of pointers, replace each Student

// in students with the Students from auxStudents in the correct order.

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

**students[k] = \*studentPtrs[k];**

} // auxStudents will be destroyed here

…