Winter 2017 CS 32

**Project 2 Solution**

In this solution, the functions with small, fast implementations are inlined. Alternatively, the inline keyword can be removed and the function implementations moved to Sequence.cpp. (inline will be mentioned at some point in class, so don't worry if you've never seen it before.)

// Sequence.h

#ifndef SEQUENCE\_INCLUDED

#define SEQUENCE\_INCLUDED

// Later in the course, we'll see that templates provide a nicer way of

// enabling us to have Sequences of different types. For now, we'll use

// a typedef.

typedef *some type* 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& other);

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

// Housekeeping functions

~Sequence();

Sequence(const Sequence& other);

Sequence& operator=(const Sequence& 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

int subsequence(const Sequence& seq1, const Sequence& 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.

void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& result);

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

// Inline implementations

inline

int Sequence::size() const

{

return m\_size;

}

inline

bool Sequence::empty() const

{

return size() == 0;

}

#endif // SEQUENCE\_INCLUDED

===================================================================

// Sequence.cpp

#include "Sequence.h"

Sequence::Sequence()

{

createEmpty();

}

Sequence::~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;

}

Sequence::Sequence(const Sequence& 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);

}

Sequence& Sequence::operator=(const Sequence& rhs)

{

if (this != &rhs)

{

Sequence temp(rhs);

swap(temp);

}

return \*this;

}

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

{

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

return false;

Node\* p = nodeAtPos(pos);

insertBefore(p, value);

return true;

}

int Sequence::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;

}

bool Sequence::erase(int pos)

{

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

return false;

Node\* p = nodeAtPos(pos);

doErase(p);

return true;

}

int Sequence::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;

}

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

{

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

return false;

Node\* p = nodeAtPos(pos);

value = p->m\_value;

return true;

}

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

{

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

return false;

Node\* p = nodeAtPos(pos);

p->m\_value = value;

return true;

}

int Sequence::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;

}

void Sequence::swap(Sequence& 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;

}

void Sequence::createEmpty()

{

m\_size = 0;

// Create dummy node

m\_head = new Node;

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

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

}

void Sequence::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++;

}

Sequence::Node\* Sequence::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;

}

Sequence::Node\* Sequence::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;

}

int subsequence(const Sequence& seq1, const Sequence& 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;

}

void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& 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 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& 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);

}