**Homework 4 Solution**

|  |  |
| --- | --- |
| [Problem 1](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/solution.html#P1) | [Problem 4](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/solution.html#P4) |
| [Problem 2](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/solution.html#P2) | [Problem 5](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/solution.html#P5) |
| [Problem 3](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/solution.html#P3) | [Problem 6](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/4/solution.html#P6) |

**Problem 1:**

// Set.h

#ifndef SET\_INCLUDED

#define SET\_INCLUDED

**template <typename ItemType>**

class Set

{

public:

Set(); // Create an empty set.

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

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

bool insert(const ItemType& value);

// Insert value into the set if it is not already present. Return

// true if the value was actually inserted. Return false if the

// value was not inserted (perhaps because it is already in the set

// or because the set has a fixed capacity and is full).

bool erase(const ItemType& value);

// Remove the value from the set if present. Return true if the

// value was removed; otherwise, leave the set unchanged and

// return false.

bool contains(const ItemType& value) const;

// Return true if the value is in the set, otherwise false.

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

// If 0 <= i < size(), copy into value an item in the set and

// return true. Otherwise, leave value unchanged and return false.

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

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

// Housekeeping functions

~Set();

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

Set**<ItemType>**& operator=(const Set**<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

struct Node

{

ItemType m\_value;

Node\* m\_next;

Node\* m\_prev;

};

Node\* m\_head;

int m\_size;

void createEmpty();

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

void insertAtTail(const ItemType& value);

// Insert value in a new Node at the tail of the list, incrementing

// m\_size.

void doErase(Node\* p);

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

Node\* find(const ItemType& value) const;

// Return pointer to Node whose m\_value == value if present, else m\_head

};

// Declarations of non-member functions

**template <typename ItemType>**

void unite(const Set**<ItemType>**& set1, const Set**<ItemType>**& set2, Set**<ItemType>**& result);

// result = { x | (x in set1) OR (x in set2) }

**template <typename ItemType>**

void subtract(const Set**<ItemType>**& set1, const Set**<ItemType>**& set2, Set**<ItemType>**& result);

// result = { x | (x in set1) AND NOT (x in set2) }

// Inline implementations

**template <typename ItemType>**

inline

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

{

return m\_size;

}

**template <typename ItemType>**

inline

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

{

return size() == 0;

}

**template <typename ItemType>**

inline

bool Set**<ItemType>**::contains(const ItemType& value) const

{

return find(value) != m\_head;

}

// Non-inline implementations

**template <typename ItemType>**

Set**<ItemType>**::Set()

{

createEmpty();

}

**template <typename ItemType>**

bool Set**<ItemType>**::insert(const ItemType& value)

{

// Fail if value already present

if (contains(value) )

return false;

// Insert new Node (at tail; choice of position is arbitrary),

// incrementing m\_size

insertAtTail(value);

return true;

}

**template <typename ItemType>**

bool Set**<ItemType>**::erase(const ItemType& value)

{

// Find the Node with the value, failing if there is none.

Node\* p = find(value);

if (p == m\_head)

return false;

// Erase the Node, decrementing m\_size

doErase(p);

return true;

}

**template <typename ItemType>**

bool Set**<ItemType>**::get(int i, ItemType& value) const

{

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

return false;

// Return the value at position i. This is one way of ensuring the

// required behavior of get: If the Set doesn't change in the interim,

// \* calling get with each i in 0 <= i < size() gets each of the

// Set elements, and

// \* calling get with the same value of i each time gets the same element.

// If i is closer to the head of the list, go forward to reach that

// position; otherwise, start from tail and go backward.

Node\* p;

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

{

p = m\_head->m\_next;

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

p = p->m\_next;

}

else // closer to tail

{

p = m\_head->m\_prev;

for (int k = m\_size-1; k != i; k--)

p = p->m\_prev;

}

value = p->m\_value;

return true;

}

**template <typename ItemType>**

void Set**<ItemType>**::swap(Set**<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>**

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

{

// 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>**

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

{

createEmpty();

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

// Inserting each new node at the tail rather than anywhere else is

// an arbitrary choice.

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

insertAtTail(p->m\_value);

}

**template <typename ItemType>**

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

{

if (this != &rhs)

{

// Copy and swap idiom

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

swap(temp);

}

return \*this;

}

**template <typename ItemType>**

void Set**<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 Set**<ItemType>**::insertAtTail(const ItemType& value)

{

// Create a new node

Node\* newNode = new Node;

newNode->m\_value = value;

// Insert new item at tail of list (predecessor of the dummy at m\_head)

// Adjust forward links

newNode->m\_next = m\_head;

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

// Adjust backward links

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

m\_head->m\_prev = newNode;

m\_size++;

}

**template <typename ItemType>**

void Set**<ItemType>**::doErase(Node\* p)

{

// 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--;

}

**template <typename ItemType>**

**typename** Set**<ItemType>**::Node\* Set**<ItemType>**::find(const ItemType& value) const

{

// Walk through the list looking for a match

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

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

;

return p;

}

**template <typename ItemType>**

void unite(const Set**<ItemType>**& s1, const Set**<ItemType>**& s2, Set**<ItemType>**& result)

{

// Check for aliasing to get correct behavior or better performance:

// If result is s1 is s2, result already is the union.

// If result is s1, insert s2's elements into result.

// If result is s2, insert s1's elements into result.

// If result is a distinct set, assign it s1's contents, then

// insert s2's elements in result, unless s2 is s1, in which

// case result now already is the union.

const Set**<ItemType>**\* sp = &s2;

if (&result == &s1)

{

if (&result == &s2)

return;

}

else if (&result == &s2)

sp = &s1;

else

{

result = s1;

if (&s1 == &s2)

return;

}

for (int k = 0; k < sp->size(); k++)

{

ItemType v;

sp->get(k, v);

result.insert(v);

}

}

**template <typename ItemType>**

void subtract(const Set**<ItemType>**& s1, const Set**<ItemType>**& s2, Set**<ItemType>**& result)

{

// Guard against the case that result is an alias for s2 by copying

// s2 to a local variable. This implementation needs no precaution

// against result being an alias for s1.

Set**<ItemType>** s2copy(s2);

result = s1;

for (int k = 0; k < s2copy.size(); k++)

{

ItemType v;

s2copy.get(k, v);

result.erase(v);

}

}

#endif // SET\_INCLUDED

**Problem 2:**

The instantiation of Set<Bottle>::insert calls Set<Bottle>::contains, which calls Set<Bottle>::find, which contains the expression p->m\_value != value, where both operands are Bottle. We never defined operator!= for Bottle operands.

**Problem 3:**

1. void listAllAuxiliary(string path, const File\* f)
2. {
3. path = path + '/' + f->name();
4. cout << path << endl;
5. if (f->files() != NULL)
6. {
7. const vector<File\*>& files = \*f->files();
8. for (size\_t k = 0; k < files.size(); k++)
9. listAllAuxiliary(path, files[k]);
10. }
11. }
12. Another way to write the for loop is
13. for (vector<File\*>::const\_iterator p = f->files()->begin();
14. p != f->files()->end(); p++)
15. listAllAuxiliary(path, \*p);
16. 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 innermost k loop. Each of the N iterations of that loop does a sequence of constant time operations (comparisons, array subscripting, addition, etc.), so that loop is O(N). Each iteration of the j loop does some constant time operations and the O(N) k loop, so the j loop is O(N2). Similarly, each of the N iterations of the i loop is O(N2), so the i loop is O(N3).
2. The innermost k loop is still O(N). The number of basic operations in the whole program is

sum(i from 0 to N-1) of [sum(j from 0 to i-1) of O(N)] ~ sum(i from 0 to N-1) of i\*O(N) ~  
O(N) \* [sum(i from 0 to N-1) of i] ~  
O(N) \* (0+1+2+3+...+(N-1)) = O(N) \* ((N-1)\*N/2) = O(N) \* ½(N2 - N) = O(N3)

For large N, this algorithm is about twice as fast than the one in part a, but it's still order N3; doubling the number of cities still increases the running time about eightfold.

**Problem 5:**

1. First, notice that our implementation of get 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).

The statements in the k loop are executed N times. Each time, the get function visits min(k,N-k) nodes (which is bounded by N), and insert visits O(N) nodes (since it checks for duplicates), so on each iteration, the body of the loop visits O(N) nodes. N \* O(N) = O(N2). Possibly assigning s1 to result in the else branch before the loop visits only O(N) nodes; the rest of that if statement is constant time. Thus, the running time is dominated by the loop: unite is O(N2).

1. Copying the items into v is O(N).  
   Sorting v is O(N log N).  
   Deleting the original result nodes is O(N).  
   Copying the nonduplicate items from v is O(N).  
   Destroying v is O(N).

Since O(N log N) dominates O(N), this version of unite is O(N log N).

**Problem 6:**

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

…

inline

bool compareStorePtr(const Store\* lhs, const Store\* rhs)

{

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

}

…

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

{

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

**{**

**Store currentStore(s[k]);**

**size\_t m = k;**

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

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

**s[m] = currentStore;**

**}**

}

…

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

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

// time.

{

vector<Store> auxStores(stores);

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

// to point to the corresponding Store in auxStores.

**vector<Store\*> storePtrs;**

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

**storePtrs.push\_back(&auxStores[k]);**

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

// with compareStorePtr as the ordering relationship.

**sort(storePtrs.begin(), storePtrs.end(), compareStorePtr);**

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

// in stores with the Stores from auxStores in the correct order.

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

**stores[k] = \*storePtrs[k];**

} // auxStores will be destroyed here

…