Hotel(const Hotel &h);

const Hotel& operator=(const Hotel &h);

~Hotel();

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

Hotel::~Hotel() {

for (int i = 0; i < num\_floors; i++) {

delete [] occupancy\_per\_floor[i];

}

delete [] occupancy\_per\_floor;

delete [] num\_rooms\_per\_floor;

}

==

Hotel::Hotel(const Hotel &h) {

num\_floors = h.num\_floors;

num\_rooms\_per\_floor = new int[num\_floors];

occupancy\_per\_floor = new bool\*[num\_floors];

for (int i = 0; i < num\_floors; i++) {

int num\_rooms = h.num\_rooms\_per\_floor[i];

num\_rooms\_per\_floor[i] = num\_rooms;

occupancy\_per\_floor[i] = new bool[num\_rooms];

for (int j = 0; j < num\_rooms; j++) {

occupancy\_per\_floor[i][j] = false;

}

} }

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

void ScrubReviews( std::list<std::vector<std::string> > &reviews, const std::string &word) {

for (std::list<std::vector<std::string> >::iterator itr = reviews.begin();

itr != reviews.end(); /\* itr incremented later \*/) {

bool erased = false;

for (int i = 0; i < itr->size(); i++) {

if ((\*itr)[i] == word) {

itr = reviews.erase(itr);

erased = true;

break;

} }

if (!erased) itr++;

}

}

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

void PrintData(Node \*head) {

if (head == NULL) return;

std::cout << head->value << " ";

PrintData(head->next);

}

void twirl (Node \*head, int value) {

assert (head != NULL);

// first, find the node that we want to twirl around

Node \*tmp = head;

while (tmp != NULL && tmp->value != value) {

tmp = tmp->next;

}

assert (tmp->value == value);

// just make sure we are aren't near either end of the links!

// (we must have at least 2 nodes before and 2 nodes after)

assert (tmp->prev != NULL);

assert (tmp->prev->prev != NULL);

assert (tmp->next != NULL);

assert (tmp->next->next != NULL);

// set up a couple of temporary pointers

Node \*new\_prev = tmp->next;

Node \*new\_next = tmp->prev;

// change the 6 links

new\_prev->prev = new\_next->prev;

new\_next->next = new\_prev->next;

new\_prev->prev->next = new\_prev;

new\_next->next->prev = new\_prev;

new\_prev->next = new\_next;

new\_next->prev = new\_prev;

// delete the twirled node

delete tmp; }

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

template <class T> Vec<T>& Vec<T>::operator=(const Vec<T>& v) {

// check for self-assignment!

if (this != &v) {

// only reallocate if the two structures have different sizes

if (m\_alloc != v.m\_alloc) {

delete [] m\_data;

this->m\_alloc = v.m\_alloc;

this->m\_data = new T[this->m\_alloc];

}

this->m\_size = v.m\_size;

// copy the data

for (unsigned int i = 0; i < this->m\_size; ++i) {

this->m\_data[i] = v.m\_data[i]; }

}

return \*this;

}

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

template <class T>

T& Vec<T>::operator[] (unsigned int i) {

return m\_data[m\_first+i];

}

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

template <class T>

void Vec<T>::push\_front(const T& val) {

// if it's the first element, just use push\_back

if (m\_alloc == 0) { push\_back(val); return; }

assert (m\_alloc > 0);

if (m\_first == 0) {

// Calculate the new allocation. Make sure it is at least one.

m\_alloc \*= 2;

assert (m\_alloc > 1);

// Allocate the new array

T\* new\_data = new T[ m\_alloc ];

// put the existing data in the back half of the array

m\_first = m\_alloc / 2;

// copy the data

for (unsigned int i=0; i<m\_size; ++i) {

new\_data[m\_first+i] = m\_data[i]; }

// delete the old array and reset the pointers

delete [] m\_data;

m\_data = new\_data;

}

// move the first index back one spot

m\_first--;

// Add the value at the last location and increment the bound

m\_data[m\_first] = val;

++ m\_size;

}

Node\* sorted\_splice(Node\* a, Node\*b) {

if (a == NULL) return b;

if (b == NULL) return a;

Node\* answer;

Node\* tmp;

Node\* rest;

if (b->value < a->value) {

// handle the case when all of list b comes before the first element of list b

answer = b;

tmp = b;

rest = a;

} else {

answer = a;

// else, use tmp to walk down a until we find the node before the splice

tmp = a;

while (tmp->ptr != NULL && tmp->ptr->value < b->value) {

tmp = tmp->ptr;

}

rest = tmp->ptr;

tmp->ptr = b;

}

// walk through the b list until the last node

while (tmp->ptr != NULL) {

tmp = tmp->ptr;

}

// attach the rest of the first list

tmp->ptr = rest;

return answer;

}

REMEMBER TO USE: CONST AND &

delete [] data;

float \* data = new float[n];

return;

draw the dynamic allocation graph

my\_list.sort(optional\_compare\_function);

std::list<int>::iterator q = s.erase(p);

std::list<int>::iterator itr = lst.begin();

for(itr = lst.begin(); itr != lst.end(); ){

if(\*itr < 0){

itr = lst.erase(itr);

itr --;  
}

else{

itr++;  
}

}