1====

int main() {

// a data structure to hold the rental history

typedef std::map<std::string,std::pair<int,std::map<std::string,int> > > history;

history data;

// read each line

std::string name, token, costume, token2;

while (std::cin >> name >> token >> costume >> token2) {

assert (token == "rents");

assert (token2 == "costume");

// find this person's data

history::iterator itr = data.find(name);

if (itr == data.end()) {

// first visit to the shop, create a new pair

std::map<std::string,int> tmp;

tmp[costume] = 1;

data.insert(std::make\_pair(name,std::make\_pair(1,tmp)));

} else {

// otherwise, increment the total & specific costume data

itr->second.first++;

itr->second.second[costume]++;

}

// output information

std::cout << name << " has been to the shop " << data[name].first

<< " time(s) and rented a " << costume << " costume "

<< data[name].second[costume] << " time(s)." << std::endl;

}

}

O(n∗(log p+log v))

2========

// helper function to clean up unneccessary structure

void destroy(TriNode \*n) {

// base case

if (n == NULL) return;

// recursively delete the children

destroy (n->left);

destroy (n->middle);

destroy (n->right);

// then delete this node

delete n;

}

// helper function

TriNode\* copy\_mirror(TriNode \*n) {

// base case

if (n == NULL) return NULL;

// create a new node on the heap

TriNode \*tmp = new TriNode(n->val);

// copy, swapping left and right

tmp->left = copy\_mirror(n->right);

tmp->middle = copy\_mirror(n->middle);

tmp->right = copy\_mirror(n->left);

return tmp;

}

// primary function

void make\_symmetric(TriNode\* n) {

// base case

if (n == NULL) return;

// clobber existing structure on right side of tree

destroy(n->right);

// replace it with a mirror image copy

n->right = copy\_mirror(n->left);

// recurse on the middle branch of the tree

make\_symmetric(n->middle);

}

3====

template <class T>

void binarytree\_to\_linkedlist(DualNode<T> \*root, DualNode<T>\* &head, DualNode<T>\* &tail) {

// base case

if (root == NULL) {

head = tail = NULL;

return; }

// temporary variables

DualNode<T> \*l\_head, \*l\_tail, \*r\_head, \*r\_tail;

// recursive calls

binarytree\_to\_linkedlist(root->leftprev,l\_head,l\_tail);

binarytree\_to\_linkedlist(root->rightnext,r\_head,r\_tail);

// the root comes first in prefix traversal

head = root;

head->leftprev = NULL;

// after that comes the left tree (if it exists)

if (l\_head == NULL) {

l\_tail = head;

} else {

head->rightnext = l\_head;

l\_head->leftprev = head;

}

// then the right tree

// make sure the tail is set appropriately!

if (r\_head == NULL) {

tail = l\_tail;

} else {

l\_tail->rightnext = r\_head;

r\_head->leftprev = l\_tail;

tail = r\_tail;

}

}

O(n)

5=====

std::set<int> data;

int num;

// read in the data, store in a set

for (int i = 0; i < n; i++) {

std::cin >> num;

data.insert(num);

}

// output directly from the set (will be sorted!)

std::set<int>::iterator itr = data.end();

while (itr != data.begin()) {

itr--;

std::cout << \*itr << " ";

}

std::cout << std::endl;

O(nlogn)

6===

std::string majority(const std::vector<std::map<std::string,std::string> > &voters,

const std::string &category) {

// determine how many votes are necessary to 'win'

int num\_voters = voters.size();

assert (num\_voters > 0);

int majority = (int)ceil((num\_voters+1)/2.0);

// a structure to store the total count for each vote

std::map<std::string,int> votes;

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

std::map<std::string,std::string>::const\_iterator itr = voters[i].find(category);

if (itr != voters[i].end()) {

votes[itr->second]++;

}

}

// go through the votes and find the max/majority

for (std::map<std::string,int>::const\_iterator itr2 = votes.begin(); itr2 != votes.end(); itr2++) {

if (itr2->second >= majority)

return itr2->first;

}

// if insufficient votes for the majority

return "(no consensus)";

}

O(v ∗ (log(c) + log(k)) + k)

7====

template <class T>

int count\_at\_level(Node<T>\* n, int level) {

if (n == NULL) return 0;

if (level == 0) return 1;

int answer = 0;

for (int i = 0; i < n->children.size(); i++) {

answer += count\_at\_level(n->children[i],level-1);

}

return answer;

}

8===

template <class T>

Node<T>\* construct\_balanced(const std::vector<T> &v) {

return construct\_balanced(v,0,v.size());

}

template <class T>

Node<T>\* construct\_balanced(const std::vector<T> &v, int first, int last) {

int size = last-first;

if (size == 0) return NULL;

// create a node for the middle element

int middle = first+size/2;

Node<T> \*answer = new Node<T>(v[middle]);

// recurse to the left and right

answer->left = construct\_balanced(v,first,middle);

if (answer->left != NULL)

answer->left->parent = answer;

answer->right = construct\_balanced(v,middle+1,last);

if (answer->right != NULL)

answer->right->parent = answer;

// return the root

return answer;

}

9====

template <class T>

void left\_swivel(Node<T>\* &input) {

assert (input != NULL && input->left != NULL);

Node<T> \*orig = input;

Node<T> \*repl = input->left;

Node<T> \*parent = input->parent;

Node<T> \*mid = input->left->right;

input = repl;

repl->parent = parent;

orig->parent = repl;

repl->right = orig;

orig->left = mid;

if (mid != NULL) mid->parent = orig;

}

template <class T>

void flatten(Node<T>\* &n) {

if (n == NULL) return;

while (n->left != NULL) { left\_swivel(n); }

flatten(n->right);

}

10====

Solution: The STL map structure stores the data sorted by key.

In contrast, data in a hash table appears to be randomly ordered

(it is ordered by the hash function evaluated for each element).

These orderings are important when we compare iteration over these

structures. The map is guaranteed to iterator over the data in sorted order.

If having the data sorted is necessary for the application, the map is

probably the better data structure choice.

Solution: An STL set discards any information about the order the

elements are inserted into the structure. An STL vector preserves this

information. For the library homework it was important to maintain the

chronological ordering of the book checkouts, so a vector is preferable to a

set. Also, a vector uses less memory than a set because the data is stored

contiguously, without pointers connecting the elements. The vector is also

more efficient for accessing the elements. The subscript operator allows

random access and iteration over a vector is much faster because there is

no pointer indirection.