CSCI-1200 Data Structures
Test 3 — Practice Problem Solutions

1 Tracking Halloween Costume Rental Data [/ 23]

In Homework 7 we maintained information about what costume each customer was *currently* renting. In this problem, we would like to track the *history* of costumes each customer has rented. Consider the input sequence of rental receipts:

```
Carol rents pirate costume
Carol rents queen costume
Ann rents squirrel costume
Carol rents pirate costume
Brian rents rockstar costume
Ann rents pirate costume
Brian rents pirate costume
Carol rents rockstar costume
Carol rents pirate costume
```

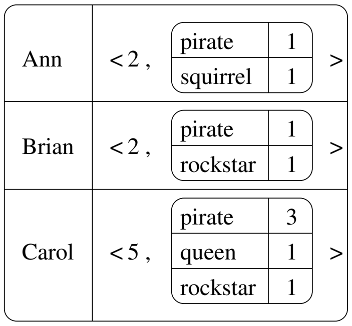
Your task is to read each line of rental information in this format and create a line of output for each rental stating the total number of times the customer has been to the shop and the number of times they rented that costume. The expected output for the above input is:

```
Carol has been to the shop 1 time(s) and rented a pirate costume 1 time(s).
Carol has been to the shop 2 time(s) and rented a queen costume 1 time(s).
Ann has been to the shop 1 time(s) and rented a squirrel costume 1 time(s).
Carol has been to the shop 3 time(s) and rented a pirate costume 2 time(s).
Brian has been to the shop 1 time(s) and rented a rockstar costume 1 time(s).
Ann has been to the shop 2 time(s) and rented a pirate costume 1 time(s).
Brian has been to the shop 2 time(s) and rented a pirate costume 1 time(s).
Carol has been to the shop 4 time(s) and rented a rockstar costume 1 time(s).
Carol has been to the shop 5 time(s) and rented a pirate costume 3 time(s).
```

Your code should be efficient for a store with many different costumes, many customers, and those customers are highly indecisive about which costume to wear.

1.1 Diagram of the costume rental history structure [/ 9]

What is the best data structure in which to store this information? Draw a picture of the rental history above stored in your selected structure using the drawing conventions from lecture. *Hint: Use STL* `map`*!*



1.2 Implementation [/ 14]

Now, finish the code below to read and process each line of input and produce each line of output. Your program will read from `std::cin` and output to `std::cout`.

Solution:

```
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;
  }
}
```

What is the order notation of your solution? Assume the input has n lines of rental information, the shop has c different costumes, there are p people/customers, each person visits the shop at most v times, and each person rents a specific costume at most k times.

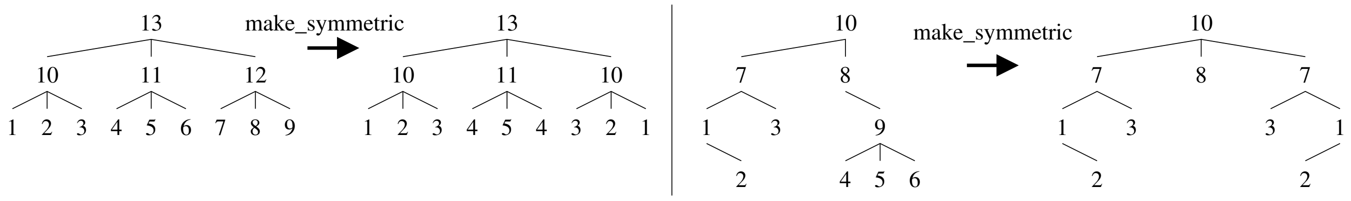
Solution: $O(n * (\log p + \log v))$ **For every line of the file, we must find the correct row of the outer map (w/ p rows), and then find the correct row of the inner map (w/ at most v rows). The two map finds are done once only per row, so they are added. By storing the total number of visits in a pair, we can avoid doing a linear sweep of the inner map to total the shop visits by that customer.**

2 Mirror Image Trinary Trees [/ 20]

In this problem you will work with trinary trees of integers, where each `TriNode` has up to 3 children. Your task is to force the structure and data to have mirror symmetry. That is, if you hold up the left half of the structure next to a mirror, the right half of the tree should be reconstructed to look like the mirror reflection. You should change the right half of the tree to look like the left side.

Here are some examples:

```
class TriNode {
public:
  TriNode(int v) : val(v),left(NULL),
    middle(NULL),right(NULL) {}
  int val;
  TriNode *left;
  TriNode *middle;
  TriNode *right;
};
```



Write a function named `make_symmetric` that takes in a single argument, a pointer to the root of a trinary tree. Your function should edit the tree as described and illustrated above. Make sure your code does not have any memory leaks or other memory errors. You'll probably want to write and use one or more helper functions in your solution. Note: You have 1.5 pages to write your answer for this problem.

Solution:

```
// 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 Converting Trees to Lists [/ 25]

In this problem you will write a recursive function named `binarytree_to_linkedlist` that takes 3 arguments of type pointer to `DualNode<T>` (defined on the right). The first function argument holds the root of a binary tree. Your function should restructure the collection of `DualNodes` to form a simple doubly-linked list of the same data in a *pre-order traversal* of the original tree.

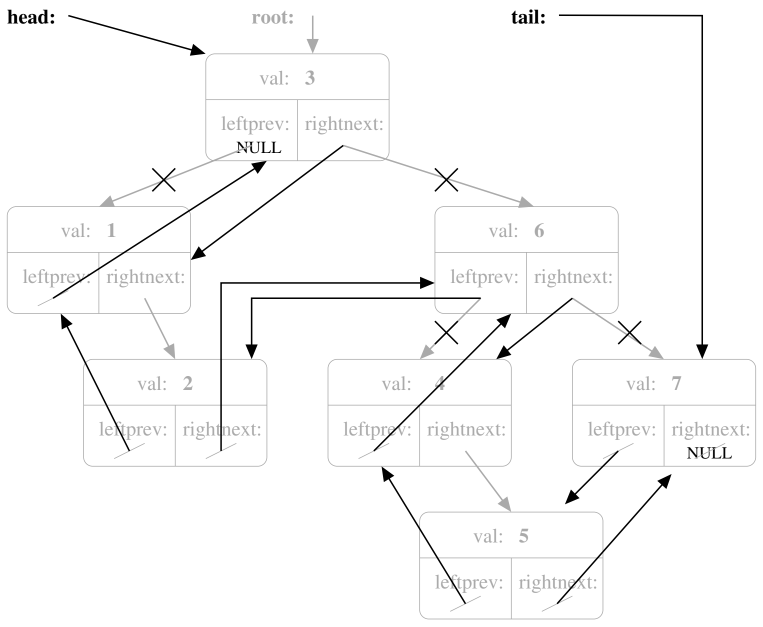
The second and third function arguments will store the head and tail, respectively, of the resulting linked list. The same two `DualNode` pointer variables within each `DualNode` are re-purposed once the data is re-structured.

Important Note: No new `DualNode`s are created by this function.

```
template <class T>
class DualNode {
public:
  DualNode<T>(const T& v) {
    val = v;
    leftprev = NULL;
    rightnext = NULL;
  }
  T val;
  DualNode<T>* leftprev;
  DualNode<T>* rightnext;
};
```

3.1 Diagram of `binarytree_to_linkedlist` [/ 8]

First, edit the diagram below to show which pointers must change to convert the structure to a doubly-linked list that is a pre-order traversal of the original tree. Please be neat!



Solution:

3.2 Implementation of `binarytree_to_linkedlist` [/ 17]

Now, implement the `binarytree_to_linkedlist` function.

Solution:

```
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;
  }
}
```

What is the order notation of your function for an input tree with n nodes and overall height h?

Solution: The function runs in $O(n)$, and the height and shape (balance/imbalance) of the structure are not relevant.

4 Tree Traversal Bingo [/ 16]

For each of the diagrams below, write a letter corresponding to one of the following statements that accurately describes the diagram. Each letter should be used exactly once.

(A) is a priority queue

(B) cannot be colored as a red-black tree

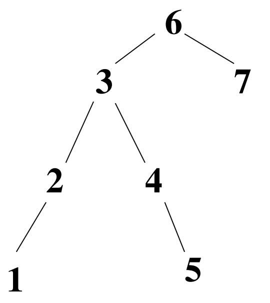
(C) has breadth-first traversal: 7 6 5 4 3 2 1

(D) is not a tree

(E) has post-order traversal: 1 2 3 4 5 6 7

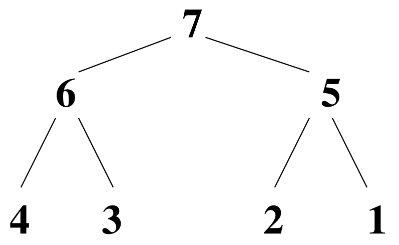
(F) is a binary search tree

4.1 [/3]



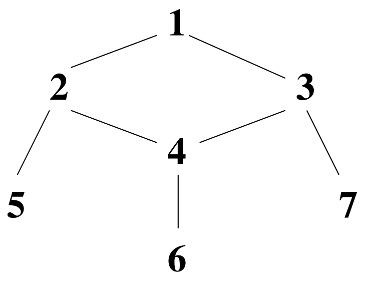
Solution: F

4.2 [/3]



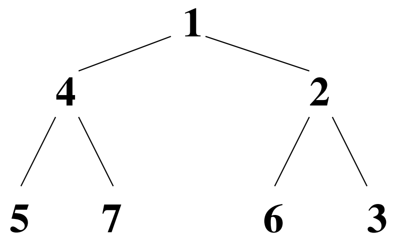
Solution: C

4.3 [/3]



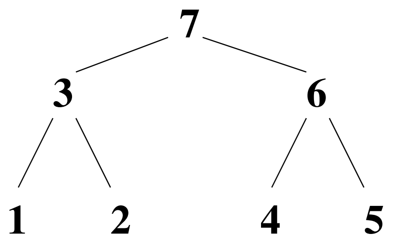
Solution: D

4.4 [/3]



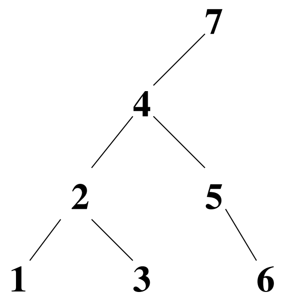
Solution: A

4.5 [/3]



Solution: E

4.6 [/3]



Solution: B (F is also correct, but F is the only answer for 4.1)

5 Sorting with a Set [/ 13]

Write a fragment of C++ code to read in n integers from `std::cin` and then output those numbers to `std::cout` in reverse sorted order (largest first). Your code should use the STL `set` container class. You may not use arrays, vectors, lists, or maps. You may assume there are no duplicates in the input data.

Solution:

```
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;
```

What is the order notation of your solution in terms of n, the number of integers read from `std::cin`? Write 2 or 3 concise and well written sentences justifying your answer.

Solution: $O(n \log n)$ **For each integer read from `std::cin` we must place that number in the set structure. Each insert take log n, so that's $n * \log n$. Printing the data is just looping/iterating (backwards) through the set structure, which takes n. $n{*}\log n + n = O(n \log n)$.**

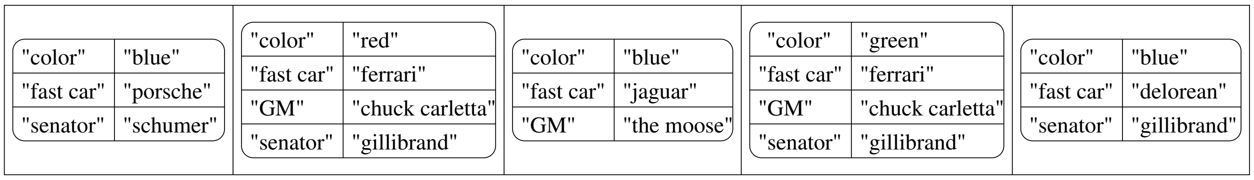
6 Majority Maps [/23]

Consider the following organization of voting data:

```
  std::map<std::string,std::string> voter_a;
    voter_a["color"] = "blue";
    voter_a["senator"] = "schumer";
    voter_a["fast car"] = "porsche";
  std::map<std::string,std::string> voter_b;
    voter_b["color"] = "red";
    voter_b["fast car"] = "ferrari";
    voter_b["GM"] = "chuck carletta";
    voter_b["senator"] = "gillibrand";
  std::map<std::string,std::string> voter_c;
    voter_c["GM"] = "the moose";
    voter_c["fast car"] = "jaguar";
    voter_c["color"] = "blue";
  std::map<std::string,std::string> voter_d;
    voter_d["color"] = "green";
    voter_d["fast car"] = "ferrari";
    voter_d["GM"] = "chuck carletta";
    voter_d["senator"] = "gillibrand";
  std::map<std::string,std::string> voter_e;
    voter_e["senator"] = "gillibrand";
    voter_e["fast car"] = "delorean";
    voter_e["color"] = "blue";
  std::vector<std::map<std::string,std::string> > voters;
  voters.push_back(voter_a); voters.push_back(voter_b); voters.push_back(voter_c);
  voters.push_back(voter_d); voters.push_back(voter_e);
```

6.1 Diagram of the `voters` structure [/8]

Draw a picture of the data stored in the `voters` variable using the conventions from lecture:



6.2 Implementing `majority` [/15]

Now, write the function `majority` to determine the 'winner' of each category. The winner must win a *majority* of the votes; that is, more than half of the voters must choose that entity. For example:

```
  std::cout << "fast car  = " << majority(voters, "fast car") << std::endl;
  std::cout << "color     = " << majority(voters, "color") << std::endl;
  std::cout << "senator   = " << majority(voters, "senator") << std::endl;
  std::cout << "GM        = " << majority(voters, "GM") << std::endl;
```

Results in this output:

```
  fast car  = (no consensus)
  color     = blue
  senator   = gillibrand
  GM        = (no consensus)
```

Make sure to use `const` and reference as appropriate in your implementation.

Solution:

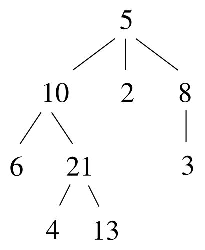
```
std::string majority(const std::vector<std::map<std::string,std::string> > &voters,
                     const std::string &category) {
  // determine how many votes are necessary to 'win'
  int num_voters = voters.size();
  assert (num_voters > 0);
  int majority = (int)ceil((num_voters+1)/2.0);
  // 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)";
}
```

What is the order notation of your solution? Assume there are v voters, c categories, k different entities in each category receiving votes, and the entity receiving the most votes receives m votes.

Solution: $O(v*(log(c)+log(k))+k)$**. Because** $k \leq v$**, we can simplify this to** $O(v*(log(c)+log(k)))$**.**

7 Counting Levels in General Trees [/16]

Write a function `count_at_level`, that takes in 2 arguments: a pointer to a `Node` and an integer `level` and returns the number of elements at that *level* of the tree. For the example to the right, we have 1 node at level 0 (the root, 5), 3 nodes at level 1, 3 nodes at level 2, and 2 nodes at level 3.



```
template <class T>
class Node {
public:
  T value;
  std::vector<Node*> children;
};
```

Solution:

```
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;
}
```

When your function is asked to count the elements on the 2nd level of the tree above, in what order does it visit the nodes of the tree? List every node that is visited in calculating the answer, not just the nodes on the 2nd level.

Solution: 5 10 6 21 2 8 3

What is the common name for the traversal order of your function: "breadth first", "bottom up", "top down", "depth first", "in order", "reverse order", "random order", or "other"?

Solution: The traversal order for this algorithm is "depth first".

8 Perfectly Balanced Binary Search Trees [/16]

Write a templated function `construct_balanced` that takes in one argument, a sorted STL `vector` of template type `T`, and returns a pointer to the root `Node` of a perfectly balanced binary search tree containing these values. You may assume the vector contains n values, where $n = 2^h - 1$, and h is the height of the final tree.

Your solution should be recursive. You may define and use a helper function as part of your solution. Make sure to correctly initialize all links between the Nodes (parent, left, and right).

```
template <class T> class Node {
public:
  Node(const T& v) :
    value(v), parent(NULL),
    left(NULL), right(NULL) {}
  T value;
  Node *parent;
  Node *left;
  Node *right;
};
```

Solution:

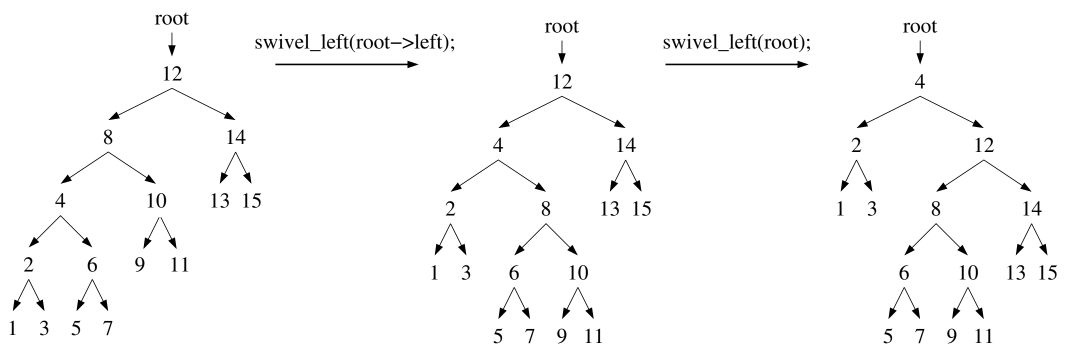
```
template <class T>
Node<T>* construct_balanced(const std::vector<T> &v) {
  return construct_balanced(v,0,v.size());
}

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 Swiveling for Balancing and Unbalancing [/28]

In this problem we will write a helper function `swivel_left` that can be used to adjust the balance/imbalance of a binary search tree structure. This templated function takes in a single argument: a pointer to the root `Node<T>`, *by reference*, and does a local re-arrangement of the nodes in the tree, so that the left node of the input Node is bumped up a level and the input node is bumped down a level. The binary search tree property of the entire tree is maintained. For example, here are two calls to the `swivel_left` function:

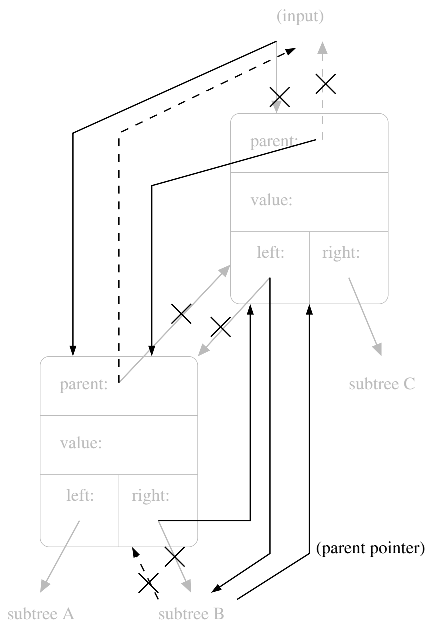
```
template <class T> class Node {
public:
  Node(const T& v) :
    value(v), parent(NULL),
    left(NULL), right(NULL) {}
  T value;
  Node *parent;
  Node *left;
  Node *right;
};
```



9.1 Diagram of `swivel_left` [/8]

First, edit the diagram below to show all of the changes that must happen to the local structure when `swivel_left` is called. Note: We will not edit the values stored in any `Node`, we will only edit the pointer variables.

Solution:



9.2 Implementation of `swivel_left` [/12]

Now, implement the `swivel_left` function. Make sure your implementation handles the corner cases where some of the pointers are NULL.

Solution:

```
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;
}
```

9.3 Converting a binary search tree into a sorted linked list [/8]

Finally, write a recursive function `flatten` that takes in the root of a binary search tree and uses `swivel_left` to restructure the binary search tree to form a completely unbalanced tree that is essentially a sorted linked list with the root pointing at the smallest element.

Solution:

```
template <class T>
void flatten(Node<T>* &n) {
  if (n == NULL) return;
  while (n->left != NULL) { left_swivel(n); }
  flatten(n->right);
}
```

10 Data Structure Comparison Short Answer [/13]

Write 3-4 concise and well-written sentences comparing the data structures below and presenting your justification for each answer.

Hash table vs. `map` [/6] We've just learned that hash tables are amazing and magically fast for find, insert, and erase operations. Why would someone choose a map over a hash table? What key feature of iteration for a `map` is not true for a hash table?

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.

`set` vs. `vector` [/7] STL `set`s ensure there are no duplicate entries in a collection of entities; for example, in HW7 a set could be used to ensure that one person did not checkout multiple copies of the same book from the library. However, in this application an STL `vector` is preferable to an STL `set`. What information cannot be represented in a `set`? What memory/performance advantages does a `vector` have over a `set`?

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.