Data structure cheat sheet
Exam 1

Lecture 14 – Associative Containers (Maps), Part 1 (and Problem Solving Too)

• STL maps store pairs of "associated" values
• Map iterators refer to pairs.
• Map search, insert and erase are all very fast: $O(\log n)$ time, where n is the number of pairs stored in the map.
• Note: The STL map type has similarities to the Python dictionary, Java HashMap, or a Perl hash, but the data structures are not the same. The organization, implementation, and performance is different. In a couple weeks we'll see an STL data structure that is even more similar to the Python dictionary.
• Map search, insert and erase are $O(\log n)$. Python dictionaries are $O(1)$.

• Maps are ordered by increasing value of the key. Therefore, there must be an operator< defined for the key.
• Once a key and its value are entered in the map, the key can't be changed. It can only be erased (together with the associated value).
• Duplicate keys can not be in the map.

• std::pair is a templated struct with just two members, called first and second.
• To work with pairs, you must #include <utility>. Note that the header file for maps (#include <map>) itself includes utility, so you don't have to include utility explicitly when you use pairs with maps.
• The function std::make_pair creates a pair object from the given values It is really just a simplified constructor, and as the example shows there are other ways of constructing pairs.
• Returning to maps, each entry in the map is a pair object of type: std::pair<const key_type, value_type>. The const is needed to ensure that the keys aren't changed! This is crucial because maps are sorted by keys!

• We've used the [] operator on vectors, which is conceptually very simple because vectors are just resizable arrays. Arrays and vectors are efficient random access data structures.
• But operator[] is actually a function call, so it can do things that aren't so simple too, for example: ++counters[s];++(counters.operator[](s)) //equivalent
• For maps, the [] operator searches the map for the pair containing the key (string)s.
– If such a pair containing the key is not there, the operator:
1. creates a pair containing the key and a default initialized value,
2. inserts the pair into the map in the appropriate position, and
3. returns a reference to the value stored in this new pair (the second component of the pair).
This second component may then be changed using operator++.
– If a pair containing the key is there, the operator simply returns a reference to the value in that pair.
• In this particular example, the result in either case is that the ++ operator increments the value associated with strings (to 1 if the string wasn't already it a pair in the map).
• For the user of the map, operator[] makes the map feel like a vector, except that indexing is based on a string (or any other key) instead of an int.
• Note that the result of using [] is that the key is ALWAYS in the map afterwards.

• Iterators may be used to access the map contents sequentially. Maps provide begin() and end() functions for accessing the bounding iterators. Map iterators have ++ and – operators.
• Each iterator refers to a pair stored in the map. Thus, given map iterator it, it->first is a const string and it->second is an int. Notice the use of " it-> ", and remember it is just shorthand for " (*it). "

• One of the problems with operator[] is that it always places a key / value pair in the map. Sometimes we don't want this and instead we just want to check if a key is there.
• The find member function of the map class does this for us. For example: m.find(key); where m is the map object and key is the search key. It returns a map iterator:
If the key is in one of the pairs stored in the map, find returns an iterator referring to this pair.
If the key is not in one of the pairs stored in the map, find returns m.end().

•The prototype for the map insert member function is:
m.insert(std::make_pair(key, value));
insert returns a pair, but not the pair we might expect. Instead it is pair of a map iterator and a bool:
std::pair<map<key_type, value_type>::iterator, bool>
• The insert function checks to see if the key being inserted is already in the map.
– If so, it does not change the value, and returns a (new) pair containing an iterator referring to the existing pair in the map and the bool value false.
– If not, it enters the pair in the map, and returns a (new) pair containing an iterator referring to the newly added pair in the map and the bool value true.

Maps provide three different versions of the erase member function:
• void erase(iterator p) — erase the pair referred to by iterator p.
• void erase(iterator first, iterator last) — erase all pairs from the map starting at first and going up to, but not including, last.
• size_type erase(const key_type& k) — erase the pair containing key k, returning either 0 or 1, depending on whether or not the key was in a pair in the map
• In C++11, the first two versions instead return an iterator pointing to the next valid element, just like vector or list erase would.

Here is an outline of the major steps to use in solving programming problems:
1. Before getting started: study the requirements, carefully!
2. Get started:
(a) What major operations are needed and how do they relate to each other as the program flows?
(b) What important data / information must be represented? How should it be represented? Consider and analyze several alternatives, thinking about the most important operations as you do so.
(c) Develop a rough sketch of the solution, and write it down. There are advantages to working on paper first. Don't start hacking right away!
3. Review: reread the requirements and examine your design. Are there major pitfalls in your design? Does everything make sense? Revise as needed.
4. Details, level 1:
(a) What major classes are needed to represent the data / information? What standard library classes can be used entirely or in part? Evaluate these based on efficiency, flexibility and ease of programming.
(b) Draft the main program, defining variables and writing function prototypes as needed.
(c) Draft the class interfaces — the member function prototypes.
These last two steps can be interchanged, depending on whether you feel the classes or the main program flow is the more crucial consideration.
5. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
6. Details, level 2:
(a) Write the details of the classes, including member functions.
(b) Write the functions called by the main program. Revise the main program as needed.
7. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
8. Testing:
(a) Test your classes and member functions. Do this separately from the rest of your program, if practical. Try to test member functions as you write them.
(b) Test your major program functions. Write separate "driver programs" for the functions if possible. Use the debugger and well-placed output statements and output functions (to print entire classes or data structures, for example).
(c) Be sure to test on small examples and boundary conditions.
The goal of testing is to incrementally figure out what works — line-by-line, class-by-class, function-by-function. When you have incrementally tested everything (and fixed mistakes), the program will work.

Lecture 15 – Associative Containers (Maps), Part 2

• A binary tree (strictly speaking, a "rooted binary tree") is either empty or is a node that has point- ers to two binary trees.
• Here's a picture of a binary tree storing integer values. In this figure,

each large box indicates a tree node, with the top rectangle representing the value stored and the two lower boxes representing pointers. Pointers that are null are shown with a slash through the box.
• The topmost node in the tree is called the root.
• The pointers from each node are called left and right. The nodes they point to are referred to as that node's (left and right) children.
• The (sub)trees pointed to by the left and right pointers at any node are called the left subtree and right subtree of that node.
• A node where both children pointers are null is called a leaf node.
• A node's parent is the unique node that points to it. Only the root has no parent.
• A binary search tree (often abbreviated to BST) is a binary tree where at each node of the tree, the value stored at the node is
– greater than or equal to all values stored in the left subtree, and
– less than or equal to all values stored in the right subtree.
• Here is a picture of a binary search tree stor- ing string values.

• The number of nodes on each subtree of each node in a "balanced" tree is approximately the same.

Lecture 16 – Trees, Part I

• STL sets are ordered containers storing unique "keys". An ordering relation on the keys, which defaults to operator<, is necessary. Because STL sets are ordered, they are technically not traditional mathematical sets.
• Sets are like maps except they have only keys, there are no associated values. Like maps, the keys are constant. This means you can't change a key while it is in the set. You must remove it, change it, and then reinsert it.
• Access to items in sets is extremely fast! O(log n), just like maps.
• Like other containers, sets have the usual constructors as well as the size member function.

Lecture 17 – Trees, Part II

• The efficiency of the main insert, find and erase algorithms depends on the height of the tree.
• The best-case and average-case heights of a binary search tree storing n nodes are both O(log n). The worst-case, which often can happen in practice, is O(n).

• Unlike binary search trees, nodes in B+ trees (and their predecessor, the B tree) have up to b children. Thus B+ trees are very flat and very wide. This is good when it is very expensive to move from one node to another.
• B+ trees are supposed to be associative (i.e. they have key-value pairs), but we will just focus on the keys.
• Just like STL map and STL set, these keys and values can be any type, but keys must have an operator< defined.
• In a B tree value-key pairs can show up anywhere in the tree, in a B+ tree all the key-value pairs are in the leaves and the non-leaf nodes contain duplicates of some keys.
• In either type of tree, all leaves are the same distance from the root.
• The keys are always sorted in a B/B+ tree node, and there are up to b - 1 of them. They act like b - 1 binary search tree nodes mashed together.
• In fact, with the exception of the root, nodes will always have between roughly b and b - 1 keys (in our 2 implementation).
• If a B+ tree node has k keys key0, key1, key2, . . . , keyk, it will have $k + 1$ children. The keys in the leftmost child must be < key0, the next child must have keys such that they are >= key0 and < key1, and so on up to the rightmost child which has only keys >= keyk.

Lecture 18 – Trees, Part III

• The increment operator should change the iterator's pointer to point to the next TreeNode in an in-order traversal — the "in-order successor" — while the decrement operator should change the iterator's pointer to point to the "in-order predecessor".
• Unlike the situation with lists and vectors, these predecessors and successors are not necessarily "nearby" (either in physical memory or by following a link) in the tree, as examples we draw in class will illustrate.
• There are two common solution approaches:
– Each node stores a parent pointer. Only the root node has a null parent pointer. [method 1]
– Each iterator maintains a stack of pointers representing the path down the tree to the current node. [method 2]
• If we choose the parent pointer method, we'll need to rewrite the insert and erase member functions to correctly adjust parent pointers.
• Although iterator increment looks expensive in the worst case for a single application of operator++, it is fairly easy to show that iterating through a tree storing n nodes requires O(n) operations overall.