CSCI-1200 Data Structures — Spring 2013
Lecture 15 – Associative Containers (Maps), Part 2

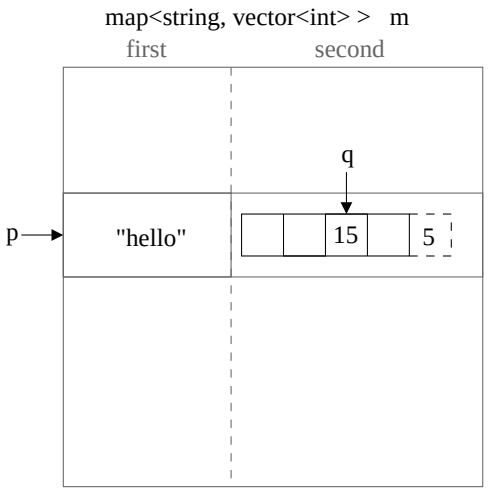
Review of Lecture 14

- Maps are associations between keys and values.
- Maps have fast insert, access and remove operations: O(log n), we'll learn why next week when we study the implementation!
- Maps store pairs; map iterators refer to these pairs.
- The primary map member functions we discussed are `operator[]`, `find`, `insert`, and `erase`.
- The choice between maps, vectors and lists is based on naturalness, ease of programming, and efficiency of the resulting program.

Today's Class — Maps, Part 2

- Maps containing more complicated values.
- Example: index mapping words to the text line numbers on which they appear.
- Maps whose keys are class objects, example: maintaining student records.
- Lists vs. Graphs vs. Trees
- Intro to Binary Trees, Binary Search Trees, & Balanced Trees

15.1 More Complicated Values



- Let's look at the example:

```
map<string, vector<int> > m;
map<string, vector<int> >::iterator p;
```

  Note that the space between the `> >` is **required**. Otherwise, `>>` is treated as an operator.

- Here's the syntax for entering the number 5 in the vector associated with the string `"hello"`:

```
m[string("hello")].push_back(5);
```

- Here's the syntax for accessing the size of the vector stored in the map pair referred to by map iterator p:

```
p = m.find(string("hello"));
p->second.size()
```

  Now, if you want to access (and change) the $\texttt{i}^{th}$ entry in this vector you can either using subscripting:

```
(p->second)[i] = 15;
```

  (the parentheses are needed because of precedence) or you can use vector iterators:

```
vector<int>::iterator q = p->second.begin() + i;
*q = 15;
```

  Both of these, of course, assume that at least `i+1` integers have been stored in the vector (either through the use of `push_back` or through construction of the vector).

- We can figure out the correct syntax for all of these by drawing pictures to visualize the contents of the map and the pairs stored in the map. We will do this during lecture, and you should do so **all the time** in practice.

15.2 Exercise

Write code to count the odd numbers stored in the map

```
map<string, vector<int> > m;
```

This will require testing all contents of each vector in the map. Try writing the code using subscripting on the vectors and then again using vector iterators.

15.3 A Word Index in a Text File

```
// Given a text file, generate an alphabetical listing of the words in the file
// and the file line numbers on which each word appears.  If a word appears on
// a line more than once, the line number is listed only once.
#include <algorithm>
#include <cctype>
#include <iostream>
#include <map>
#include <string>
#include <vector>
using namespace std;

// implementation omitted, will be covered in a later lecture
vector<string> breakup_line_into_strings(const string& line);

int main() {
  map<string, vector<int> > words_to_lines;
  string line;
  int line_number = 0;

  while (getline(cin, line)) {
    line_number++;
    // Break the string up into words
    vector<string> words = breakup_line_into_strings(line);

    // Find if each word is already in the map.
    for (vector<string>::iterator p = words.begin(); p!= words.end(); ++p) {
      // If not, create a new entry with an empty vector (default) and
      // add to index to the end of the vector
      map<string, vector<int> >::iterator map_itr = words_to_lines.find(*p);
      if (map_itr == words_to_lines.end())
        words_to_lines[*p].push_back(line_number);  // could use insert here
      // If it is, check the last entry to see if the line number is
      // already there.  If not, add it to the back of the vector.
      else if (map_itr->second.back() != line_number)
        map_itr->second.push_back(line_number);
    }
  }

  // Output each word on a single line, followed by the line numbers.
  map<string, vector<int> >::iterator map_itr;
  for (map_itr = words_to_lines.begin(); map_itr != words_to_lines.end(); map_itr++) {
    cout << map_itr->first << ":\t";
    for (unsigned int i = 0; i < map_itr->second.size(); ++i)
      cout << (map_itr->second)[ i ] << " ";
    cout << "\n";
  }
  return 0;
}
```

15.4 Our Own Class as the Map Key

- So far we have used `string` (mostly) and `int` (once) as the key in building a `map`. Intuitively, it would seem that `string` is used quite commonly.
- More generally, we can use any class we want as long as it has an `operator<` defined on it.
- Suppose we want to maintain data for students including name, address, courses, grades, and tuition fees and calculate things like GPAs, credits, and remaining required courses. We could do this by making a single Student class object that stores everything for a particular student and put that in a vector or list. Alternately, we could break the information into separate classes and use a map. First, let's look at a sketch of a few classes that can work together to store the data:

```
class Name {
public:
  Name(const string& first, const string& last) :
    m_first(first), m_last(last) {}
  const string& first() const { return m_first; }
  const string& last() const { return m_last; }

private:
  string m_first;
  string m_last;
};


class CourseGrade {
public:
  Course(const string &c_name, const string & grade) : course_name(c_name), final_grade(grade) {}
  const string & get_course_name() const { return course_name; }
  const string & get_final_grade() const { return final_grade; }

private:
  string course_name;
  string final_grade;
};


class StudentRecord {
public:
  const string& getAddress() const { return address; }
  const string& getGradeInCourse(const string &course_name) const; /* implementation omitted */
  bool hasCompletedCourse(const string &course_name) const; /* implementation omitted */
  float getGPA() const { return GPA; }
  /* additional member functions omitted */

private:
  string address;
  vector<CourseGrade> completed_coursework;
  float GPA;
  /* etc. */
};
```

- Now if we want to create a map of student names and associated student records, we need to add an `operator<` for `Name` objects. This is simple:

```
bool operator< (const Name& left, const Name& right) {
  return left.last() < right.last() ||
    (left.last() == right.last() && left.first() < right.first());
}
```

- Now we can define a map:

```
map<Name, StudentRecord> students;
```

15.5 Exercises

- First let's draw a picture of this map data structure populated with interesting data:

- So what are the advantages of organizing this data using a map in this way? Let's assume there are s students, c different classes offered at the school, each student takes up to k classes before graduation, and at most p students take a particular course.
  - Write a fragment of code to access student X's grade in course Y. What is the order notation of this operation?

  - Write a fragment of code to make a list of *all* students who have taken course Y. What is the order notation of this operation?

15.6 Typedefs

- One of the painful aspects of using maps is the syntax. For example, consider a constant iterator in a map associating strings and vectors of ints:

```
map < string, vector<int> > :: const_iterator p;
```

- Typedefs are a syntactic means of shortening this. For example, if you place the line:

```
typedef map < string, vector<int> > map_vect;
```

  before your main function (and any function prototypes), then anywhere you want the map you can just use the identifier `map_vect`:

```
map_vect :: const_iterator p;
```

  The compiler makes the substitution for you.

15.7 When to Use Maps, Reprise

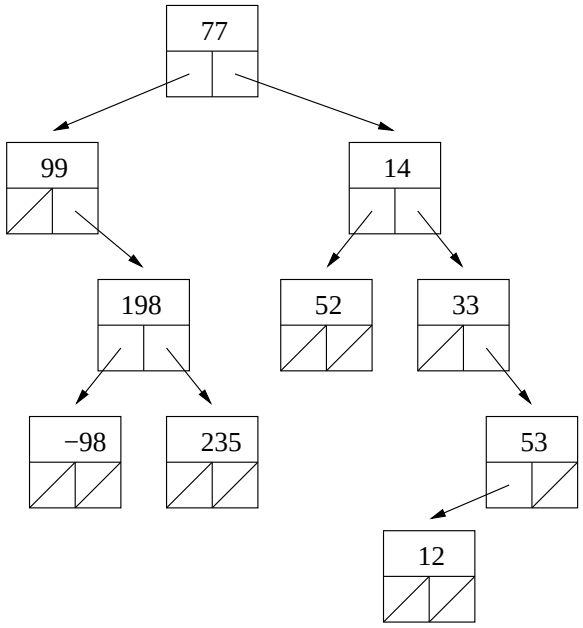
- Maps are an association between two types, one of which (the key) must have a `operator<` ordering on it.
- The association may be immediate:
  - Words and their counts.
  - Words and the lines on which they appear
- Or, the association may be created by splitting a type:
  - Splitting off the name (or student id) from rest of student record.

15.8 Overview: Lists vs. Trees vs. Graphs

- Trees create a hierarchical organization of data, rather than the linear organization in linked lists (and arrays and vectors).
- Binary search trees are the mechanism underlying maps & sets (and multimaps & multisets).
- Mathematically speaking: A *graph* is a set of vertices connected by edges. And a tree is a special graph that has no *cycles*. The edges that connect nodes in trees and graphs may be *directed* or *undirected*.

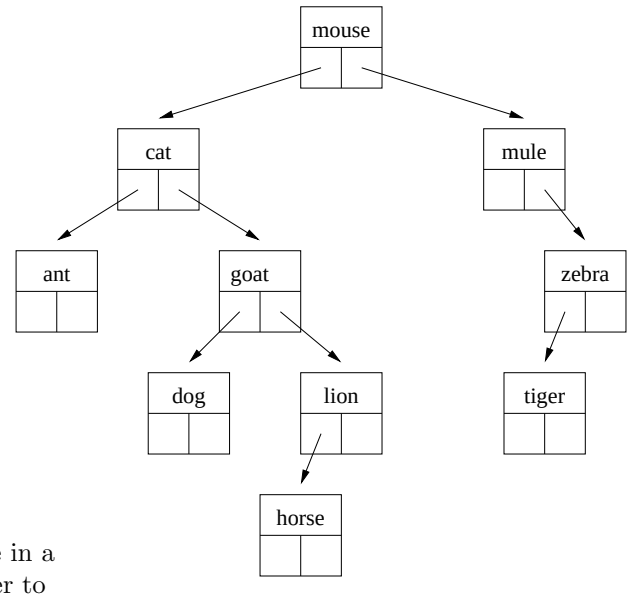
15.9 Definition: Binary Trees

- A binary tree (strictly speaking, a "rooted binary tree") is either empty or is a node that has pointers 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.



15.10 Definition: Binary Search Trees

- A *binary search tree* 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 storing string values.



15.11 Definition: Balanced Trees

- The number of nodes on each subtree of each node in a "balanced" tree is *approximately* the same. In order to be an *exactly* balanced binary tree, what must be true about the number of nodes in the tree?
- In order to claim the performance advantages of trees, we must assume and ensure that our data structure remains approximately balanced. (You'll see much more of this in Intro to Algorithms!)

15.12 Exercise

Consider the following values:

```
4.5, 9.8, 3.5, 13.6, 19.2, 7.4, 11.7
```

1. Draw a binary tree with these values that *is NOT* a binary search tree.

2. Draw *two different* binary search trees with these values. Important note: This shows that the binary search tree structure for a given set of values is not unique!

3. How many *exactly balanced* **binary search trees** exist with these numbers? How many *exactly balanced* **binary trees** exist with these numbers?