CSCI-1200 Data Structures — Spring 2016
Lecture 17 — Introduction to Sets & Problem Solving Techniques, Continued

Announcements: Test 2 Information

- Test 2 will be held **Monday, April. 4, 2016 from 6-7:50pm**.
- Students will be randomly assigned to a test room (DCC 308 or DCC 318) and seating zone. Check the homework submission website on Monday.
- No make-ups will be given except for pre-approved absence or illness, and a written excuse from the Dean of Students or the Student Experience office or the RPI Health Center will be required.
- If you have a letter from the Student Experience Office for extra exam time you should have received an e-mail from Prof. Thompson indicating the location and time for the test. Contact the professor by email **TODAY Friday April 1 if you have a letter from the Student Experience office and have not receiced this e-mail.**
- Coverage: Lectures 1-17, Labs 1-9, and Homeworks 1-7. Trees will not be included.
- A link to practice problems from previous exams is posted on the calendar `http://www.cs.rpi.edu/academics/courses/spring16/csci1200/calendar.php`. Solutions will be posted on the course website on Saturday. The best way to prepare is to completely work through and write out your solution to the problems, *before* looking at the answers.
- UPE will hold and Exam #2 Review Session on Friday 4/1/16 at 4pm in Lally 104.
- Bring to the exam room:

  Your Rensselaer photo ID card.

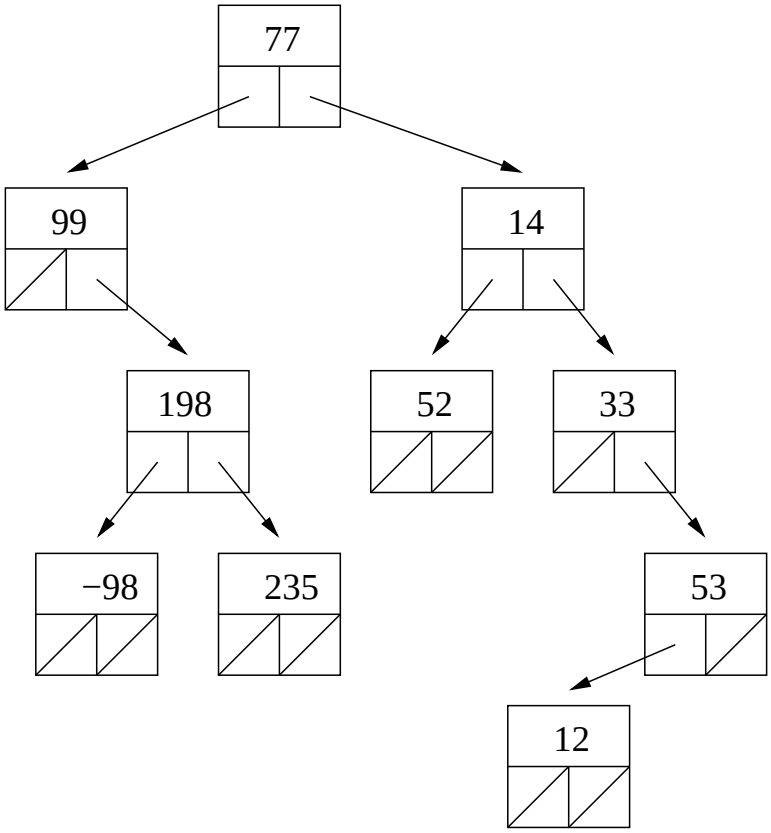
  Pencil(s) & eraser (pens are ok, but not recommended). The exam *will* involve handwritten code on paper (and other short answer problem solving). Neat legible handwriting is appreciated. We will be somewhat forgiving to minor syntax errors – it will be graded by humans not computers :)

  [OPTIONAL] You may bring 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed. Learn how to print double-sided. You may not staple or tape or glue together 2 or more single-sided sheets of paper.
- Do not bring your own scratch paper. We will provide scratch paper.
- Computers, cell-phones, smart watches, calculators, music players, etc. are not permitted. Please do not bring your laptop, books, backpack, etc. to the exam room – leave everything in your dorm room. *Unless you are coming directly from another class or sports/club meeting.*

Review of Lecture 16

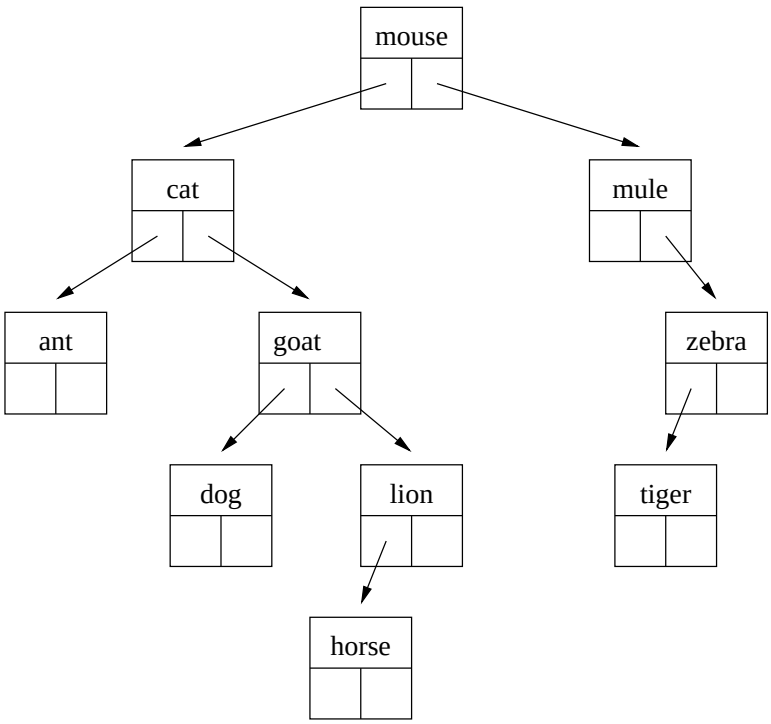
- Maps can contain complex structres such as vectors, lists, or other maps as values.
- A binary tree is either empty or is a node that has pointers to two binary trees.
- The topmost node in the tree is called the *root*.
- The pointers from each node are called *left* and *right*. The nodes they point 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.

Here's an example of a binary tree



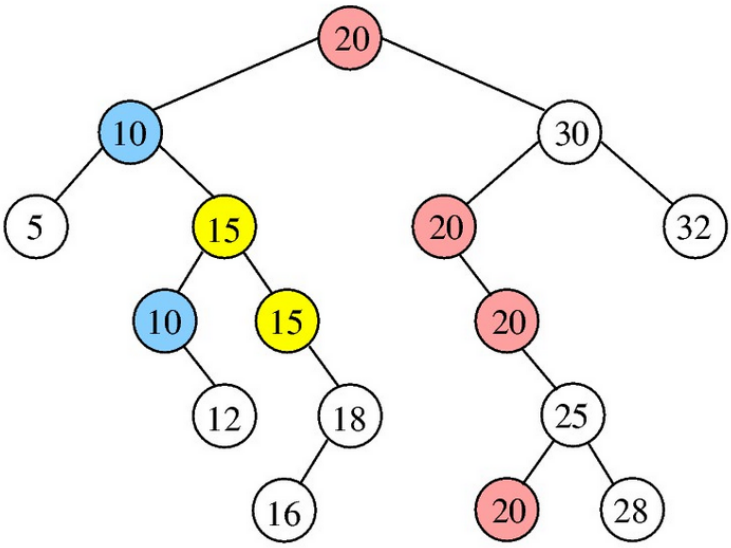
15.1 Definition: Binary Search Trees

- 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.



- Many definitions of a binary tree specify that duplicate nodes are not allowed.

- The binary trees underlying maps and sets do not allow duplicates.
- Duplicate nodes complicate search algorithm, so are not always allowed.
- Duplicate nodes are sometimes stored as linked lists in nodes with duplicate values.
- Values at all nodes in the left subtree of u $<$Value at $u \leq$Values at all nodes in the right subtree of **u**
- Here is a picture of a binary search tree storing duplicate values.



15.2 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!)

15.3 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!

Today's Lecture

- Introduction to Sets
- A couple of programmming problem examples
  - Design Example: The Game of Life
  - Quicksort

17.1 Standard Library Sets

- 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.

17.2 Set iterators

- Set iterators, similar to map iterators, are bidirectional: they allow you to step forward (`++`) and backward (`--`) through the set. Sets provide `begin()` and `end()` iterators to delimit the bounds of the set.
- Set iterators refer to const keys (as opposed to the pairs referred to by map iterators). For example, the following code outputs all strings in the set `words`:

```
for (set<string>::iterator p = words.begin(); p!= words.end(); ++p)
  cout << *p << endl;
```

17.3 Set `insert`

- There are two different versions of the `insert` member function. The first version inserts the entry into the set and returns a pair. The first component of the returned pair refers to the location in the set containing the entry. The second component is true if the entry wasn't already in the set and therefore was inserted. It is false otherwise. The second version also inserts the key if it is not already there. The iterator `pos` is a "hint" as to where to put it. This makes the insert faster if the hint is good.

```
pair<iterator,bool> set<Key>::insert(const Key& entry);
iterator set<Key>::insert(iterator pos, const Key& entry);
```

17.4 Set `erase`

- There are three versions of `erase`. The first `erase` returns the number of entries removed (either 0 or 1). The second and third erase functions are just like the corresponding erase functions for maps. Note that the `erase` functions do not return iterators. This is different from the `vector` and `list` erase functions.

```
size_type set<Key>::erase(const Key& x);
void set<Key>::erase(iterator p);
void set<Key>::erase(iterator first, iterator last);
```

17.5 Set `find`

- The find function returns the `end` iterator if the key is not in the set:

```
const_iterator set<Key>::find(const Key& x) const;
```

17.6 Design Example: Conway's Game of Life

Let's design a program to simulate Conway's Game of Life. Initially, due to time constraints, we will focus on the main data structures of needed to solve the problem.

Here is an overview of the Game:

- We have an infinite two-dimensional grid of cells, which can grow arbitrarily large in any direction.

- We will simulate the life & death of cells on the grid through a sequence of generations.
- In each generation, each cell is either alive or dead.
- At the start of a generation, a cell that was dead in the previous generation becomes alive if it had exactly 3 live cells among its 8 possible neighbors in the previous generation.
- At the start of a generation, a cell that was alive in the previous generation remains alive if and only if it had either 2 or 3 live cells among its 8 possible neighbors in the previous generation.
  - With fewer than 2 neighbors, it dies of "loneliness".
  - With more than 3 neighbors, it dies of "overcrowding".
- Important note: all births & deaths occur simultaneously in all cells at the start of a generation.
- Other birth / death rules are possible, but these have proven to be a very interesting balance.
- Many online resources are available with simulation applets, patterns, and history. For example:
  http://www.math.com/students/wonders/life/life.html
  http://www.radicaleye.com/lifepage/patterns/contents.html
  http://www.bitstorm.org/gameoflife/
  http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Applying the Problem Solving Strategies

In class we will brainstorm about how to write a simulation of the Game of Life, focusing on the representation of the grid and on the actual birth and death processes.

Understanding the Requirements

We have already been working toward understanding the requirements. This effort includes playing with small examples by hand to understand the nature of the game, and a preliminary outline of the major issues.

Getting Started

- What are the important operations?
- How do we organize the operations to form the flow of control for the main program?
- What data/information do we need to represent?
- What will be the main challenges for this implementation?

Details

- New Classes? Which STL classes will be useful?

Testing

- Test Cases?

17.7 Example: Quicksort

- Quicksort also the partition-exchange sort is another efficient sorting algorithm. Like mergesort, it is a divide and conquer algorithm.
- Quicksort first divides a large array into two smaller sub-arrays, the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.
- The steps are:
  1. Pick an element, called a pivot, from the array.
  2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
  3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- Here's an example of a quicksort routine. Let's compare it to mergesort.

```
#include <iostream>
#include <vector>
using namespace std;

int quickSort(vector<double>& array, int start, int end);


int partition(vector<double>& array, int start, int end, int& swaps) {
  int mid = (start + end)/2;
  double pivot = array[mid];


  }
}


int quickSort(vector<double>& array, int start, int end) {
  int swaps = 0;
  if(start < end) {
    int pIndex = partition(array, start, end, swaps);

    //after each call one number(the PIVOT) will be at its final position
    swaps += quickSort(array, start, pIndex-1);
    swaps += quickSort(array, pIndex+1, end);
  }

  return swaps;
}


int main() {
  vector<double> pts(7);
  pts[0] = -45.0;  pts[1] =   89.0;  pts[2] =    34.7;  pts[3] =   21.1;
  pts[4] =   5.0;  pts[5] =  -19.0;  pts[6] =  -100.3;

  quickSort(pts, 0, pts.size()-1);

  for (unsigned int i=0; i<pts.size(); ++i)
    cout << i << ": " << pts[i] << endl;
}
```