

lecture01: Intro, Pointers, and Memory

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

10th June, 2013

Introductions

Welcome to CS 225!

- Me: Sean Massung (massung1)
 - Grad student in CS, undergrad here as well
 - Interested in information retrieval, natural language processing (and CS education!)
- Course Staff:
 - Tom Bogue (tbogue2)
 - Jason Cho (hcho33)
 - Joe Ciurej (ciurej2)
 - Chase Geigle (geigle1)
- You...

What do you expect to get out of this class?

CS 225 101

- Please read the syllabus!
- ...and visit the home page.
- We'll go over any course infrastructure questions *tomorrow*

Week one is the busiest week

- Linux tutorial tomorrow night, Tuesday, June 11th at 7pm
- hw0 is out, due Wednesday, June 12th at 11:59pm in SVN
 - What's SVN? How do I turn it in? Everything is explained in hw0 and the syllabus.
 - Make note: hw0 **must** be typed!
- Labs meet tomorrow (9am or 11am)
- mp1 released Wednesday (due Monday, June 17th)
- *Please remember to read the syllabus before tomorrow's lecture*

Anything but “Hello World”

```
/** @file main.cpp */
#include <iostream>
using namespace std;

int main(int argc, char** args)
{
    int n = 47;
    cout << "My favorite number is " << n << endl;
    return 0;
}
```

- Compiles with `g++ main.cpp -o myprog`
- Execute with `./myprog`
- What happens?

Variables

- Each variable you declare has its own special place in memory

```
int num = 225;  
char grade = 'A';  
double gpa = -6.8;
```

addr	name	value	type
0x700	num	225	int
0x704	grade	A	char
0x705	gpa	-6.8	double

- Pointer** types refer to another variable's location in memory

```
int* pnum = &num;  
double* pgpa = &gpa;
```

0x70d	pnum	0x700	int*
0x711	pgpa	0x705	double*

How to use pointers?

- **Dereference** a pointer with the `*` operator to access the pointee

```
*pgpa; // -6.8
```

- **&**, or the **address-of** operator, shows where its argument resides in memory

```
&grade; // 0x704
```

- Change a pointee value:

```
*pgpa = 5.0;
```

```
cout << gpa << endl; // prints "5.0"
```

addr	name	value	type
0x700	num	225	int
0x704	grade	A	char
0x705	gpa	-6.8 5.0	double
0x70d	pnum	0x700	int*
0x711	pgpa	0x705	double*

Stack memory vs heap memory

■ Stack memory

- ...is what we've used so far
 -
 - When variables on the stack “go out of scope”, they
 - Relatively size compared to the heap

■ Heap memory

- We'll explain how to use it on the next slide
 - Memory is allocated with `new` and `delete`
 - Note the `delete` function — memory on the heap must be explicitly freed by the user or else...
 -
 - Both are stored in RAM, just in separate areas

Pointer madness!

```
// 1
int x = 9;
int* ptr;
ptr = &x;
int** pptr = &ptr;
int* optr = ptr;

// 2
*optr = 47;

// 3
ptr = new int;
*ptr = 100;
int* fav = new int(7);
```

Stack Memory

addr	name	value	type
0x716	x	9 47	int
0x71a	ptr	0x716 0xc00	int*
0x71e	pptr	0x71a	int**
0x721	optr	0x716	int*
0x725	fav	0xc04	int*

Heap Memory

addr	name	value	type
0xc00	-	100	int
0xc04	-	7	int

Graphical pointer representation

```
// 1
int x = 9;
int* ptr;
ptr = &x;
int** pptr = &ptr;
int* optr = ptr;
```

Graphical pointer representation

```
// 1
int x = 9;
int* ptr;
ptr = &x;
int** pptr = &ptr;
int* optr = ptr;

// 2
*optr = 47;
```

Graphical pointer representation

```
// 1
int x = 9;
int* ptr;
ptr = &x;
int** pptr = &ptr;
int* optr = ptr;

// 2
*optr = 47;

// 3
ptr = new int;
*ptr = 100;
int* fav = new int(7);
```

How does this make you feel?

Can you draw what is happening?

```
// 1
double* radius = new double(6.2);
double half = *radius / 2.0;
double* phalf = &half;
cout << "sum: " << *phalf + *radius << endl;
```

```
// 2
int* thing;
*thing = 77;
```

```
// 3
int* thing = NULL;
*thing = 47;
cout << "Can you see me?" << endl;
```

How about this?

Compiler errors:

```
int* ptr = &47;  
int* ptr = 34;
```

Runtime errors:

```
int* myval;  
int* otherval = new int(1);  
int val = *myval + *otherval;
```

Pointers can be tricky, even for experienced programmers

- A paper describes finding 56 pointer related bugs in a Linux file system implementation (see *Defective Error/Pointer Interactions in the Linux Kernel*)

Make sure you can draw a picture when dealing with pointers!
(drawing a picture is actually general advice you should carry with you throughout the rest of this class and your future ones)

lecture02: Memory, Arrays, and Parameters

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

11th June, 2013

Announcements

- hw0 due tomorrow night (6/12)
- Linux tutorial tonight in the lab
- mp1 released tomorrow night (due Monday, 6/17)
- lab_lab due Thursday night (6/13)

Day Two

- How was lab_lab?
- Did you read the syllabus?
Questions/clarifications?
- Last time: pointers and
memory
- This time: pointers,
memory, arrays, and
parameters
- You can do it!



Pointer Quiz

```
double* ptr = new double(0.0);
double** dptr = new double*(&ptr);
```

How should we free the memory we allocated?

- How many deletes do we need?
- Does the order matter?

```
int** dptr = new int*;
*dptr = new int(47);
```

How should we free the memory we allocated?

- How many deletes do we need?
- Does the order matter?

Are there errors?

```
// 1
size_t length = 47;
int* plength = &length;

// 2
double stars = 4.3;
double* pstars = stars;

// 3
int*** nickels = new int**(new int*(new int(5)));
***nickels = ***nickels * ***nickels;
cout << ***nickels << endl;
```

One more time

```
// 1
int* p;
int x = 5;
p = &x;
delete x;
p = NULL;

// 2
int *mine, *yours; // note this declaration
mine = new int;
yours = mine;
*yours = 8;
delete yours;
*mine = 12;
mine = NULL;
```

Time to Celebrate Arrays

So far, we've only dealt with single items. But you should all know that we can make *arrays*, too!

- Just like variables before, arrays can be on the stack or the heap
- You **must** know the size of the array at compile time if the array lives on the

```
int arr[5]; // stack or heap?  
for(size_t i = 0; i < 5; ++i)  
    arr[i] = 0;
```

A very useful program

```
int main(int argc, char** argv)
{
    // assume we have a get_size() function
    size_t length = get_size();

    int* arr = new int[length];
    for(size_t i = 0; i < length; ++i)
        arr[i] = 0;

    return 0;
}
```

Are we done?

2D array on the stack

```
size_t w = 10, h = 12;  
// w and h are known at compile-time  
double arr[w][h];  
  
for(size_t i = 0; i < w; ++i)  
    for(size_t j = 0; j < h; ++j)  
        arr[i][j] = 1.0;
```

2D array on the heap

```
double** arr = new double*[w];  
  
for(size_t i = 0; i < w; ++i)  
    arr[i] = new double[h];  
  
for(size_t i = 0; i < w; ++i)  
    for(size_t j = 0; j < h; ++j)  
        arr[i][j] = 1.0;  
  
for(size_t i = 0; i < w; ++i)  
    delete [] arr[i];  
delete [] arr;
```

A point to ponder

How many ways can we allocate memory for `cell`?

```
int* cell;
```

```
// 1
```

```
cell =
```

```
// 2
```

```
cell =
```

Counting the cell

```
int** cell;  
// 1  
cell =
```

```
// 2  
cell =
```

```
// 3  
cell =
```

```
// 4  
cell =
```

Passing variables

- So far, we've only written code inside the main function
- In reality, we want to write functions in order to decompose our logic
 - In fact, in CS 242, all functions have to be less than 35 lines
 - r/badcode
- Warning for you C programmers: this will be slightly different than what you know. Also, we will *never* use `void*` parameters or return types

Baby's first functions

```
// this function takes an int by value and returns nothing
void nothing(int x) {
    x = 99;
}

// this function has no parameters and returns an int
int number() {
    return 47;
}

int main() {
    int y = number();
    nothing(y);
    cout << "y: " << y << endl;
}
```

Baby's first functions

```
// this function takes an int* by value and returns nothing
void something(int* x) {
    *x = 99;
}

// this function has no parameters and returns an int
int number() {
    return 47;
}

int main() {
    int y = number();
    something(    );    // ?
    cout << "y: " << y << endl;
}
```

Baby's first functions

```
// this function takes an int by reference and returns nothing
void something(int & x) {
    x = 99;
}

// this function has no parameters and returns an int
int number() {
    return 47;
}

int main() {
    int y = number();
    something(y);
    cout << "y: " << y << endl;
}
```

Now you can do an exam question!

```
void kiwi(int x) {  
    x *= 2;  
}  
  
void pear(int* x) {  
    *x *= 3;  
}  
  
void durian(int & x) {  
    x *= 4;  
}  
  
/*  
 * What is printed?  
 */  
int main() {  
    int x = 1;  
    kiwi(x);  
    pear(&x);  
    durian(x);  
    cout << x << endl;  
    return 0;  
}
```

Three (really two) ways to pass variables

```
/**  
 * Passing by value  
 * makes a copy of  
 * the object  
 */
```

```
void kiwi(int a)  
{  
    a = 1;  
}  
  
void main()  
{  
    int x = 0;  
    kiwi(x);  
}
```

```
/**  
 * You can also  
 * pass a pointer  
 * (by value) to  
 * the object  
 */
```

```
void pear(int* a)  
{  
    *a = 2;  
}
```

```
void main()  
{  
    int x = 0;  
    pear(&x);  
}
```

```
/**  
 * Passing by  
 * reference gives  
 * the variable a  
 * different name  
 */
```

```
void durian(int & a)  
{  
    a = 3;  
}
```

```
void main()  
{  
    int x = 0;  
    durian(x);  
}
```

How is this useful?

- When we talk about objects tomorrow, it will be much more obvious why (e.g.) passing by reference is useful
- When passing by reference (or by “pointer”), we don’t have to
 - . This makes our code more *efficient*
- Passing by reference is nice, because we never have to check for ! It is harder to be passed an invalid reference than an invalid pointer

Const

Const is a keyword (look, it's green!). It is a modifier, which means it is added onto other type names to change them. Once assigned, a const variable's data

```
int x = 34;  
const double pi = 3.141592635897932384626;
```

```
// of course, this is OK  
x = 23;
```

```
// compiler error! pi is const, so it can't be modified  
pi = 22.0 / 7;
```

Wikipedia snippet: const and pointers

```
void foo(int * ptr,
          int const * ptrToConst,
          int * const constPtr,
          int const * const constPtrToConst)
{
    *ptr = 0;
    ptr = NULL;
    *ptrToConst = 0;
    ptrToConst = NULL;
    *constPtr = 0;
    constPtr = NULL;
    *constPtrToConst = 0;
    constPtrToConst = NULL;
}
```

Equivalent syntax

```
void foo(int * ptr,  
         const int * ptrToConst,  
         int * const constPtr,  
         const int * const constPtrToConst)  
{  
    *ptr = 0;  
    ptr = NULL;  
    *ptrToConst = 0;  
    ptrToConst = NULL;  
    *constPtr = 0;  
    constPtr = NULL;  
    *constPtrToConst = 0;  
    constPtrToConst = NULL;  
}
```

Const return values

Of course, return values can be `const` as well:

```
const int* get_ptr() {
    const int* x = new int(47);
    return x;
}

int main() {
    const int* ptr = get_ptr();
    --(*ptr); // Error! modifies the pointee
    delete ptr;

    return 0;
}
```

Const correctness

- **Const correctness** is a programming methodology that tries to enhance program correctness and readability by declaring variables (and as we'll see later, functions), as `const`
- If you *know* a variable should not be changed, mark it as `const`. This enhances your code's maintainability
- That way, if someone (maybe not you!) decides to modify it, the action will result in a compiler error

lecture03: Classes and Compilation

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

12th June, 2013

Announcements

- hw0 due tonight (6/12)
- lab_lab “due” tomorrow night (6/13)
- lab_debug tomorrow morning
- Is anyone going to office hours?

Intro to OOP

- Object-oriented programming (OOP) is a large component of this course: it allows us to
- C++ is an object-oriented programming language since it has support for user-defined types
- This allows us to write much more maintainable code; `int` and `double` are nice, but they are not very expressive!

Using an object in C++

```
/**  
 * @file main.cpp  
 */  
#include <iostream>  
#include "sphere.h"  
  
int main(int argc, char** argv)  
{  
    Sphere s;  
    s.setRadius(2.4);  
    std::cout << s.getRadius() << std::endl;  
  
    return 0;  
}
```

Writing an object in C++

```
/** @file sphere.h */\n\n\nclass Sphere\n{\n    public:\n        void setRadius(double newRadius);\n        double getRadius() const;\n\n    private:\n        double radius;\n\n};\n// note the semicolon above!
```

The cpp file

```
/**  
 * @file sphere.cpp  
 */  
#include "sphere.h"  
  
Sphere::setRadius(double newRadius)  
{  
    // maybe do some value checking here...  
    radius = newRadius;  
}  
  
Sphere::getRadius() const  
{  
    return radius;  
}
```

The whole Sphere class

```
/**  
 * @file sphere.h  
 */  
  
#ifndef SPHERE_H  
#define SPHERE_H  
  
class Sphere  
{  
public:  
    void setRadius(double newRadius);  
    double getRadius() const;  
  
private:  
    double radius;  
};  
  
#endif  
  
/**  
 * @file sphere.cpp  
 */  
  
#include "sphere.h"  
  
Sphere::setRadius(double newRadius)  
{  
    radius = newRadius;  
    // or...  
  
    // or...  
}  
  
Sphere::getRadius() const  
{  
    return radius;  
}
```

Recap so far

- Objects in C++ consist of two files: the header (.h) and implementation (.cpp). Why?
- Which code is the “client” concerned with?
- public and private fields
- const on class functions
- The scope operator, ::
- The arrow operator, ->
- The this keyword: a pointer to the current object

The compilation process

- We want to create an executable file from a collection of source files
- First, the source files (.h and .cpp) are turned into ending with the extension **.o**
 - Object files contain compiled code that is not directly runnable
- Then, the object files are input to the **linker**, which generates an executable—for example, `myprog`—which can be run as `./myprog`

Compiling our example

- 1 Compile main.cpp into an object file, main.o

```
g++ -c main.cpp
```

- 2 Compile sphere.cpp into sphere.o, another object file

```
g++ -c sphere.cpp
```

- 3 Link main.o and sphere.o to create an executable

```
g++ main.o sphere.o -o myprog
```

- 4 Run it!

```
./myprog
```

The exact commands for these steps is not necessary to know now. In general, you can count on the given Makefile to perform the correct operations.

Pointers and classes

```
Sphere* s = new Sphere;  
s->setRadius(4.7);           // remember that arrow thing?  
(*s).setRadius(4.8);
```

```
Car* c;  
c.drive(); // ok?  
c->drive(); // ok?
```

```
Mystery m;  
Mystery* mptra = &m;  
mptr->something();
```

Arrays and classes

```
Loudspeaker speakers[4];
for(size_t i = 0; i < 4; ++i)
    speakers[i].play();
```

```
Loudspeaker* speakers = new Loudspeaker[num_speakers];
for(size_t i = 0; i < num_speakers; ++i)
    speakers[i].play();
```

```
Loudspeaker** guess = // ??
```

Functions and classes

```
// any potential issues here?  
print_radius(Sphere s)  
{  
    std::cout << "The radius is: "  
    std::cout << s.getRadius() << std::endl;  
}  
  
void main()  
{  
    Sphere s;  
    s.setRadius(99.0);  
    print_radius(s);  
}
```

A better radius printer

```
// any potential issues here?  
print_radius(Sphere* s)  
{  
    std::cout << "The radius is: "  
    std::cout << s->getRadius() << std::endl;  
}  
  
void main()  
{  
    Sphere s;  
    s.setRadius(99.0);  
    print_radius(&s);  
}
```

A much better radius printer

```
// pass by const reference!
print_radius(const Sphere & s)
{
    std::cout << "The radius is: ";
    std::cout << s.getRadius() << std::endl;
}

void main()
{
    Sphere s;
    s.setRadius(99.0);
    print_radius(s);
}
```

Reading a header file

```
class SquareMaze {  
public:  
    /**  
     * @param width The width of the SquareMaze (number of cells)  
     * @param height The height of the SquareMaze (number of cells)  
     */  
    void makeMaze(int width, int height);  
  
    /**  
     * @param x The x coordinate of the current cell  
     * @param y The y coordinate of the current cell  
     * @param dir The desired direction to move from the current cell  
     * @return whether you can travel in the specified direction  
     */  
    bool canTravel(int x, int y, int dir) const;  
  
    /**  
     * @return a PNG of the unsolved SquareMaze  
     */  
    PNG drawMaze() const;
```

lecture04: Constructors and Destructors

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

13th June, 2013

Announcements

- lab_debug due Saturday night (6/15)
- mp1 due Monday night (6/17)

Warmup: what happens?

```
/** @file main.cpp */

void changeGrade(Student stu)
{
    if(stu.getGrade() == 'F')
        stu.setGrade('A');
}

void main()
{
    Student s;
    s.setGrade('F');
    while(s.getGrade() == 'F')
        changeGrade(s);
}
```

```
/** @file student.h */

#ifndef STUDENT_H
#define STUDENT_H

class Student
{
public:
    void setGrade(char letter);
    char getGrade() const;

private:
    char grade;
};

#endif
```

Motivation

- **Constructors** are a mechanism for running a function during
- They are not explicitly called; rather,
- If no constructors are written, a default no-argument constructor is

```
Sphere s(4.0);  
Student joe("Joe Ciurej", 'A');  
Student chase("Chase Geigle", 'D');  
Student tom("Tom Bogue", 'Q');
```

Writing constructors

```
/** @file sphere.h */

#ifndef SPHERE_H
#define SPHERE_H

class Sphere
{
public:
    Sphere(double newRadius);
    void setRadius(double newRadius);

private:
    double radius;
};

#endif
```

```
/** @file sphere.cpp */

#include "sphere.h"

Sphere::Sphere(double newRadius)
{
    radius = newRadius;
}

Sphere::setRadius(double newRadius)
{
    radius = newRadius;
}
```

A shortcut for constructors

Usually, constructors just set a bunch of variables. Therefore, there is a succinct syntax called an **initializer list** that does just that:

```
// one-parameter constructor
Sphere::Sphere(double newRadius): radius(newRadius) { /* nothing */ }

// "default" constructor sets the radius to zero
Sphere::Sphere(): radius(0.0) { /* nothing */ }

// this constructor takes two arguments
Sphere::Sphere(double newRadius, const std::string & newColor):
    radius(newRadius), color(newColor) { /* nothing */ }
```

Note that we still need the braces after the function declaration even if there are no statements in the function.

Course staff is questionable

What should the two-parameter constructor look like to enable the following functionality of the Student class?

```
Student jason("Jason Cho", 'F');
```

Can you write both implementations: initializer list and function body? Hint: "Jason Cho" is a string object.

Notes on constructors

- If you write any constructor, the system no longer provides a default constructor
- If you do not set member variables in the constructor, their values will be uninitialized
- ...so you should *always*

Motivation: what happens here?

```
/** @file main.cpp */
#include "sphere.h"

void main()
{
    Sphere s(4.0);

    // or...

    Sphere* s = new Sphere(4.0);
    delete s;
}
```

```
/** @file sphere.h */
#ifndef SPHERE_H
#define SPHERE_H

class Sphere
{
public:
    Sphere(double newRadius);
private:
    double* radius;
};

#endif

/** @file sphere.cpp */
#include "sphere.h"

Sphere::Sphere(double newRadius)
{
    radius = new double(newRadius);
}
```

Our first destructor

```
/** @file sphere.h */

#ifndef SPHERE_H
#define SPHERE_H

class Sphere
{
public:
    Sphere(double newRadius);
    ~Sphere();

private:
    double* radius;
};

#endif
```

```
/** @file sphere.cpp */

#include "sphere.h"

Sphere::Sphere(double newRadius)
{
    radius = new double(newRadius);
}

Sphere::~Sphere()
{
    delete radius;
}
```

Destructors, *not* “deconstructors”

- Destructors are the opposite of constructors: they are run when an object on the goes out of scope or when an object on the is deleted
- Destructors are not explicitly called by the user; the system calls them as necessary
- Think of it this way: calling `delete` on a pointer to an object on the heap *invokes* the destructor
- If your object allocates any , then you need a destructor!

The Collage class

```
/** @file collage.h */
#ifndef COLLAGE_H
#define COLLAGE_H

#include <iostream>
#include "png.h"
using namespace std;

class Collage
{
public:
    Collage(size_t numPics);
    ~Collage();

private:
    PNG* pics;
};

#endif
```

```
/** @file collage.cpp */

#include "collage.h"

Collage::Collage(size_t numPics)
{
    cout << "Created!" << endl;
    pics = new PNG[numPics];
}

Collage::~Collage()
{
    cout << "Destruacted!" << endl;
    delete [] pics;
}
```

What is printed in each of these functions?

```
void onStack()
{
    cout << "Creating A..." << endl;
    Collage colA(24);

    cout << "Creating B..." << endl;
    Collage colB(12);
}

void onHeap()
{
    Collage* col = new Collage(4);
    delete col;
}
```

Bonus: RAII

- Resource Acquisition is Initialization (RAII) is a C++ idiom
- The constructor acquires a resource, and the destructor frees it
- Useful for writing exception-safe C++ code
- Examples:
 -
 -
 -
 -

lecture05: The Big Three

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

17th June, 2013

Announcements

- mp1 due tonight at 11pm
- mp2 released tonight
 - Significantly longer than mp1!
 - Make sure to start early and use the provided test cases
- lab_memory tomorrow morning



This class's class

```
/** @file book.h */

// from now on, let's omit
// inclusion guards for brevity

#include <string>
using std::string;

class Book
{
public:
    Book(const string & title);
    ~Book();

private:
    string _title;
    string* _lines;
};
```

```
/** @file book.cpp */

#include "book.h"

Book::Book(const string & title):
    _title(title)
{
}

Book::~Book()
{
}

// note: it's common practice
// to prefix private member
// variables with an underscore
```

Using a Book

// We can use the book class as we expect

```
Book mybook("Speaker for the Dead");
```

```
Book* other = new Book("Children of the Mind");  
delete other;
```

// What if we want to make a copy of a book?

```
Book mycopy(mybook);
```

// the object mycopy is a copy of "Speaker for the Dead"

You wouldn't copy a book

There's a problem with calling the *implicit copy constructor* of our Book class. The system-provided "copy constructor" essentially performs the following steps:

```
Book mycopy;  
mycopy._title = mybook._title; // ok?  
mycopy._lines = mybook._lines; // ok?
```

By default, a
picture of what happens?

is made! Can you draw a

Announcements

Review

The Copy Constructor

The Assignment Operator

Making a shallow copy of a Book

I would copy a book

- In order to fix this problem, we need to
- This is similar to creating our own constructor to override the default system one
- Our copy constructor will copy over all member variables from a parameter object to the current object, ensuring
- Let's check out our modified Book class...

Adding the copy constructor

```
/** @file book.h */

#include <string>
using std::string;

class Book
{
public:
    Book(const string & title);
    Book(const Book & other);
    ~Book();

private:
    string _title;
    string* _lines;
};


```

```
/** @file book.cpp */
#include "book.h"

Book::Book(const string & title):
    _title(title)
{
    _lines = new string[100];
}

Book::Book(const Book & other) {
    _title = other._title;
    _lines = new string[100];
    for(size_t i = 0; i < 100; ++i)
}

Book::~Book() {
    delete [] _lines;
}
```

Points to ponder

- Now that we defined our own copy constructor, each book created with it has its own memory
- Why pass in `other` by reference?
- Why is `other` a `const` variable?
- Remember, by default *all objects* have a default shallow-copy copy constructor!

What else can we do?

// So now we can safely do this:

```
Book another("Shadow of the Hegemon");  
Book same(another);
```

// But what about this?

```
Book another("Shadow of the Hegemon");  
Book same = another;
```

*// Are the same issues encountered with =
// as with the copy constructor?*

Operator overloading

- Yes! By default, `operator=` in C++ makes a shallow copy
- Fortunately, in C++ we can to make
`operator=` do exactly what we want
- In a very similar way to the copy constructor, we can add an
`operator=` method to our Book class
- It's a class member because it can also be written as
`same.operator=(another);`
- Think about it as `lhs.operator=(rhs);`

Adding operator=

```
/** @file book.h */

#include <string>
using std::string;

class Book {

public:

Book(const string & title);
Book(const Book & other);
Book & operator=(const Book & rhs);
~Book();

private:

string _title;
string* _lines;

};
```

```
/** @file book.cpp */

Book & Book::operator=(const Book & rhs)
{
    // protect against re-assignment
    // clear lhs

    // copy rhs
    // return a helpful value
}
```

Don't duplicate your code!!

```
/** @file book.h */

class Book {

public:

    Book(const string & title);
    Book(const Book & other);
    Book & operator=(const Book & rhs);
    ~Book();

private:

    string _title;
    string* _lines;

    void copy(const Book & other);
    void clear();

};
```

```
/** @file book.cpp */

Book::Book(const Book & other)
{
}

Book & Book::operator=(const Book & rhs)
{
    if(this != &rhs) {

    }

    return *this;
}

Book::~Book()
{
}
```

Things to think about

- Why not if(*this != rhs)?
- Why return a Book &?

So why was this lecture called “The Big Three”?

- Any guesses what “the big three” are in C++? Why?
- **If you have any reason to implement one of the big three, you must implement all of them!**
- The above item was bold, it must be important
- Disclaimer: the rule of three becomes the rule of five in C++11 (which we don’t use in this class). Just be aware!

lecture06: Inheritance

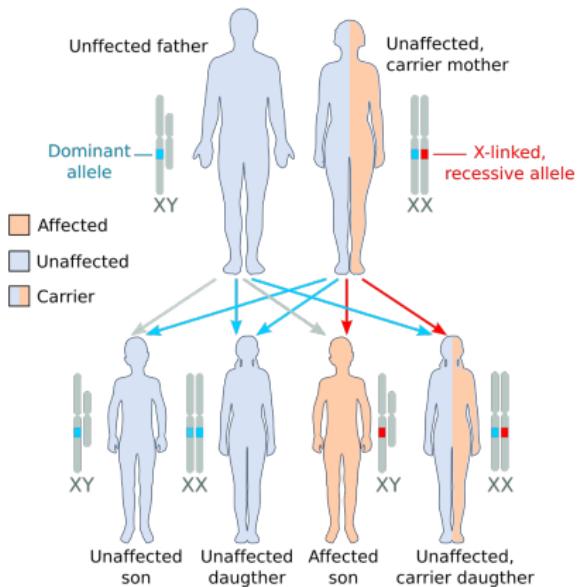
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

18th June, 2013

Announcements

- mp2 released last night
- lab_memory due Thursday evening (6/20)
- mp2.1 extra credit due Friday evening (6/21)



The Animal class

```
/** @file animal.h */

#include <string>
using std::string;

class Animal
{
public:
    Animal(const string & name);
    void eat();
    void sleep() const;
    void speak() const;

private:
    // ...
};
```

```
/** @file main.cpp */

Animal ani("Fred");
ani.sleep();
ani.speak(); // "????"

// what if you want:

Lion lion("George");
lion.sleep();
lion.speak(); // "roar!"

Penguin pen("Awkward");
pen.eat();
pen.speak(); // "ork ork!"

// all animals share the same sleep
// and eat functions but have
// different speak functions
```

Base and derived classes (subclass/superclass)

```
/** @file animal.h */

#include <string>
using std::string;

class Animal
{
public:
    Animal(const string & name);
    void eat();
    void sleep() const;
    void speak() const;

private:
    // ...
};
```

```
/** @file lion.h */
#include "animal.h"

class Lion: public Animal
{
public:
    Lion(const string & name);
    void speak() const;
};

/** @file penguin.h */
#include "animal.h"

class Penguin: public Animal
{
public:
    Penguin(const string & name);
    void speak() const;
};
```

Inheritance

```
/** @file penguin.h */

#include <iostream>
#include "animal.h"

using namespace std;

// a Penguin "is an" Animal
class Penguin: public Animal
{
public:
    Penguin(const string & name);
    void speak() const;
};
```

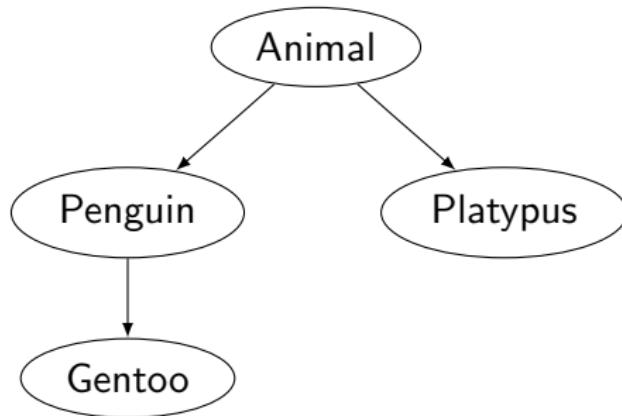
```
/** @file penguin.cpp */

// this constructor calls the base
// class constructor with the
// Animal's name argument
Penguin(const string & name):
    Animal(name)
{
    // initialize the Penguin
}

void Penguin::speak() const
{
    cout << "ork ork!" << endl;
}

// the sleep and eat functions are
// inherited from the Animal class
// so we don't have to write them
```

Inheritance diagram



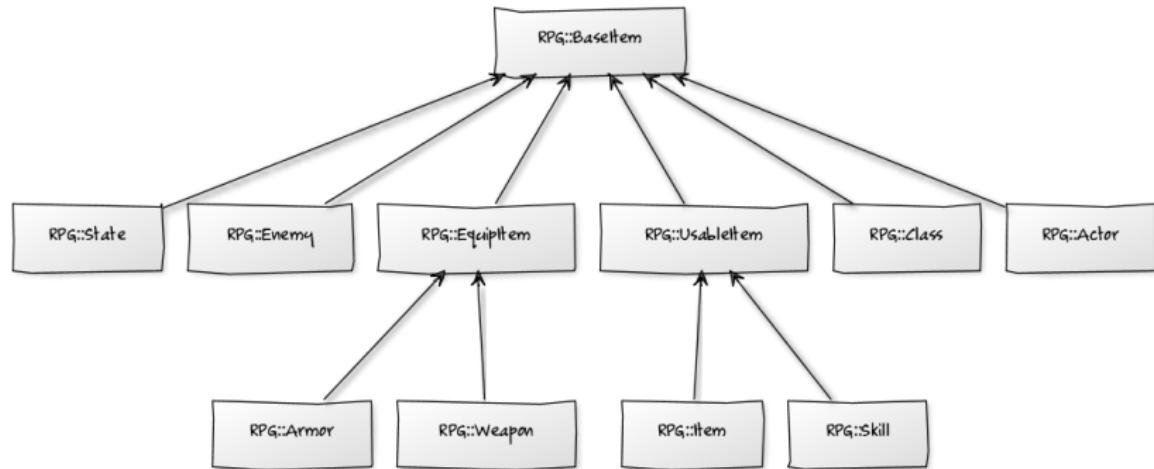
*/** @file main.cpp */*

```
Penguin pen("Todd");  
pen.speak();  
pen.eat();
```

```
Platypus plat("Perry");  
plat.speak();  
plat.sleep();
```

```
Gentoo gent("Ted");  
gent.sleep();
```

Inheritance diagram for an RPG



Source: <http://cobbtocs.co.uk/wp/wp-content/uploads/2012/04/>

Inheritance nuances

- Base private members are inaccessible to derived class members
 - Use the `protected` field to allow this, but still disallow client code from accessing them
- We will only discuss “public inheritance” in class; that is, `class Derived: public Base`
 - This means public functions in the base remain public in the derived class

OOP

- We've now talked about two of the three main attributes of object oriented programming
- Previously, we discussed
- We just covered
- We will also talk about
- Wouldn't this be a great interview question for a software engineering intern?

How do these make you feel?

```
Animal ani("Francis");
Penguin pen("Zoe");
ani = pen;
pen = ani;
```

```
Animal* ani;
Penguin* pen;
ani = pen;
pen = ani;
```

What happens here?

```
Animal ani("Francis");
Penguin pen("Zoe");
ani.speak();
pen.speak();
```

```
Animal* ani = new Animal("Peter");
ani->speak();
```

```
Animal* ani = new Penguin("Claire");
ani->speak();
```

Static vs dynamic binding

```
Animal* ani = new Penguin("Claire");  
ani->speak(); // ???
```

Why didn't this one print out "ork ork!"? Answer: C++ has static (or "early") binding by default. We want dynamic binding; that is, we want to be able to look up the correct function to call at runtime:

```
Animal* ani;  
if(/* some condition */)  
    ani = new Penguin("Linda");  
else  
    ani = new Platypus("Doug");  
ani->speak();
```

Enabling dynamic binding

To enable dynamic binding for a particular function, we use the keyword `virtual` in the function definition:

```
class Animal
{
public:
    Animal(const string & name);
    void eat();
    void sleep() const;
    virtual void speak() const;
};
```

This means any deriving class that implements `speak` will have its own code called. Remember, this only becomes an issue if pointers are being used.

Dynamic binding is cool!

```
// make a heterogeneous array

Animal** zoo = new Animal*[4];

zoo[0] = new Donkey("Jason");
zoo[1] = new Fish("Joe");
zoo[2] = new Chameleon("Chase");
zoo[3] = new Camel("Tom");

for(size_t i = 0; i < 4; ++i)
```

Dynamic binding is really cool!

```
// ...or make a function whose output
// is determined at runtime

void animal_speak(Animal* animal)
{
}

void main()
{
    Animal* one = new Giraffe("Jason");
    Animal* two = new Hippo("Joe");
    animal_speak(one);
    animal_speak(two);
}
```

Virtual function facts

- A virtual method is one a derived class can
- A derived class is not required to override an existing implementation of an inherited virtual method
- Constructors can't be
- Destructors *should* be virtual. Why?

Virtual destructors: what is printed?

```
class Base {  
public:  
    Base() { cout << "ctor: B" << endl; }  
    ~Base() { cout << "dtor: B" << endl; }  
};  
  
class Derived: public Base {  
public:  
    Derived() { cout << "ctor: D" << endl; }  
    ~Derived() { cout << "dtor: D" << endl; }  
};  
  
void main() {  
    Base* bptr = new Derived;  
    delete bptr;  
}
```

Virtual destructors: now what is printed?

```
class Base {  
public:  
    Base() { cout << "ctor: B" << endl; }  
    virtual ~Base() { cout << "dtor: B" << endl; }  
};  
  
class Derived: public Base {  
public:  
    Derived() { cout << "ctor: D" << endl; }  
    ~Derived() { cout << "dtor: D" << endl; }  
};  
  
void main() {  
    Base* bptr = new Derived;  
    delete bptr;  
}
```

Concluding remarks

- Polymorphism: objects of different types can employ methods of the
- Inheritance provides
- Now you know the three major components of OOP!
- Finishing up dynamic polymorphism tomorrow with *abstract base classes*
- Then: static polymorphism via *templates*

lecture07: Templates

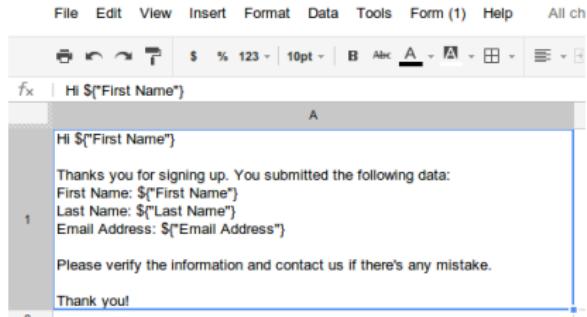
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

19th June, 2013

Announcements

- lab_memory due tomorrow evening (6/20)
- mp2.1 due Friday evening (6/21)
- mp2 due Monday evening (6/24)



Source: https://developers.google.com/apps-script/images/mail_merge_image5.png

Finishing up a concept from yesterday

- What if a function is declared `virtual`, but never implemented?
- The class becomes an `abstract base class`, and cannot be instantiated (think “interface”)
- Let’s turn our `Animal` base class into an abstract base class by making `speak` be
- Instead of making the default `Animal` say “????”, we’ll make it so you can’t create one

An abstract base class

```
/** @file animal.h */

#include <string>
using std::string;

class Animal
{
public:
    Animal(const string & name);
    void eat();
    void sleep() const;
    // note the = 0 below:
    virtual void speak() const = 0;
};

// an abstract base class has at
// least one pure virtual function
```

```
/** @file main.cpp */

// compiler error!
Animal ani("Fred");

Lion lion("George");
lion.sleep();
lion.speak(); // "roar!"

Animal* pen = new Penguin("Awkward");
pen->eat();
pen->speak(); // "ork ork!"
```

Finishing up dynamic polymorphism

- Derived classes must implement all pure virtual functions; otherwise the deriving class will also be abstract and can't be instantiated
- An abstract base class contains a list of functions that must be implemented; this way, we can be sure that derived classes have some specific functionality

Some motivation

```
void swap_int(int & a, int & b)
{
    int temp = a;
    a = b;
    b = temp;
}

void swap_sphere(Sphere & a, Sphere & b)
{
    Sphere temp = a;
    a = b;
    b = temp;
}

void swap_double(double & a, double & b)
{
    double temp = a;
    a = b;
    b = temp;
}
```

```
// do we really need this many
// swap functions??

int main()
{
    int x = 42;
    int y = 47;
    Sphere s1(4.0);
    Sphere s2(6.0);
    double a = 0.01;
    double b = 0.9;

    swap_int(x, y);
    swap_sphere(s1, s2);
    swap_double(a, b);

    return 0;
}
```

Don't duplicate your code!!

- Clearly, all this duplication is bad; the only difference in the functions was the type of the variables
- Can we somehow parameterize the type as well as the variables?
- Templates to the rescue!
- (Could we do this with dynamic polymorphism? How?)

Templates to the rescue

```
template <class T>
void swap(T & a, T & b)
{
    T temp = a;
    a = b;
    b = temp;
}

// this is certainly shorter,
// and it works for types we
// haven't even seen yet!
```

```
int main()
{
    int x = 42;
    int y = 47;
    Sphere s1(4.0);
    Sphere s2(6.0);
    double a = 0.01;
    double b = 0.9;

    swap(x, y);
    swap(s1, s2);
    swap(a, b);

    return 0;
}
```

How does this actually work?

- At compile time, the compiler finds all calls to the templated function, and replaces the templated types (in our case, T), with the actual types
- That means that a separate copy of the function is created for each type it is called with
- Since all this happens at compile time, this is an example of
- Can you write a generic `max` function?

Max function

```
template <class T>
T max(const T & a, const T & b)
{
```

```
}
```

// what is required of type T here?

Write your own “pow” function

We want the following functionality:

```
double d = 2.0;  
int i = 3;  
  
pow(d, 3);    // 8.0  
pow(d, 4);    // 16.0  
pow(i, 0);    // 1  
pow(i, 10);   // 59049
```

Can you write the templated pow function? To simplify the function, assume that the exponent is a non-negative integer.

Pow function

```
template <class T>
T pow(T base, int exp)
{
}
```

Classes can be templated, too!

A templated class can also have one or more placeholder types associated with it:

```
/** @file pair.h */

template <class T>
class Pair
{
public:
    Pair(const T & first, const T & second);
    T max() const;
private:
    T _first;
    T _second;
};
```

Using a templated class

```
/** @file pair.h */

#ifndef PAIR_H
#define PAIR_H

template <class T>
class Pair
{
public:
    Pair(const T & first,
          const T & second);
    T max() const;
private:
    T _first;
    T _second;
};

#endif
```

```
/** @file main.cpp */

#include <iostream>
#include "pair.h"
#include "sphere.h"

using namespace std;

void main()
{
    Pair<int> mypair(4, 5);
    cout << mypair.max() << endl;

    Sphere a(6.7);
    Sphere b(3.2);
    Pair<Sphere> myspheres(a, b);
    cout << myspheres.max() << endl;
}
```

Implementing a templated class

```
/** @file pair.h */

#ifndef PAIR_H
#define PAIR_H

template <class T>
class Pair
{
public:
    Pair(const T & first,
          const T & second);
    T max() const;
private:
    T _first;
    T _second;
};

#include "pair.cpp"
#endif
```

```
/** @file pair.cpp */
// no #include here

template <class T>
Pair<T>::Pair(const T & first,
               const T & second)
{
    _first = first;
    _second = second;
}

template <class T>
T Pair<T>::max() const
{
    if(_first > _second)
        return _first;
    return _second;
}
```

What do you notice?

- We include pair.cpp inside pair.h!
 - The definition of a template function or class *must* be available to the calling code
 - Otherwise,
- When scoping to the class in the .cpp file, we include the template parameter: `Pair<T>::__`
- Also inside the .cpp file, we need a template `<class T>` before each function

Pop quiz!

- Write the function signature for the copy constructor of the templated `Pair` class
- Dynamically allocate an array of size `size` of `Pairs` of `Spheres`
- Write the class definition of a template `Pair` class that takes two *different* types

Disclaimer

- `std::swap` and `std::pair` already exist
- Don't rewrite them!
- `#include <algorithm>` for `std::swap`
- `#include <utility>` for `std::pair`
- `pow` also already exists...
- What are some other potentially useful template classes or functions?

lecture08: Containers and Generic Programming

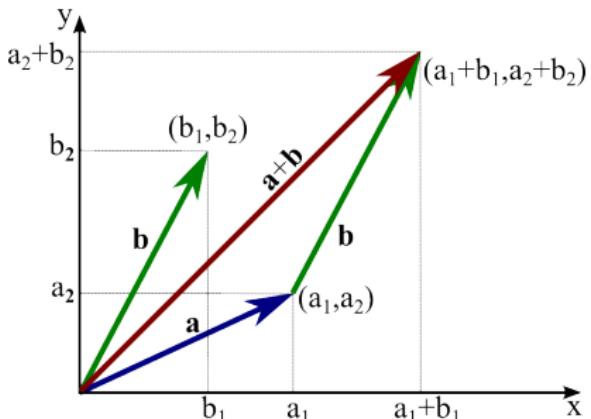
Sean Massung

CS 225 UIUC

20th June, 2013

Announcements

- lab_memory due tonight
- mp2.1 extra credit due Friday evening (6/21)
- lab_inheritance due Saturday evening (6/22)
- mp2 due Monday evening (6/24)
- mt1 Tuesday evening at 7pm



Our first data structure

- Let's implement our first *Abstract Data Type* (ADT)
 - An ADT is a list of rules describing functionality of a data structure; it prescribes functions that must be implemented
 - Sounds like an abstract base class, huh?
- We'll implement the Vector ADT with a resizing array
 - A sedan has a set functionality: four doors, and seats four to five passengers
- Think about this: an array is an implementation of a Vector like a Honda Civic is an implementation of a sedan

But what is a Vector?

Here's the Vector ADT:

- 1 Insert a new element at the back of the Vector
- 2 Remove the element at the back of the Vector
- 3 Access any element by an index
- 4 Find the size of the Vector
- 5 Check if the Vector is empty

Not only should all these functions exist, but they should be relatively efficient. Additionally, our Vector should be able to hold any type!

Lecture 8 and finally a data structure!

```
template <class T>
class Vector {
public:
    Vector();
    Vector(const Vector & other);
    Vector & operator=(const Vector & rhs);
    ~Vector();

    void push_back(const T & elem);
    T pop_back();
    T at(size_t idx) const;
    size_t size() const;
    bool empty() const;

private:
    size_t _size;
    size_t _capacity;
    T* _data;
    void copy(const Vector & other);
    void clear();
};
```

Vector constructor and push_back

```
template <class T>
Vector<T>::Vector(): _size(0), _capacity(16)
{
    _data = new T[_capacity];
}

// can you write the big 3?

template <class T>
void Vector<T>::push_back(const T & elem)
{
    _data[_size] = elem;
    ++_size;
    if(_size == _capacity)
    {
        // resize and copy elements
        // set new capacity (how?)
    }
}
```

Vector pop_back and size

```
template <class T>
T Vector<T>::pop_back()
{
}

template <class T>
size_t Vector<T>::size() const
{
}
```

Our first data structure analysis

With all data structures in CS 225, we want to analyze the running times of the ADT functions. If n is the number of elements in the `Vector`, fill in the running times with Big- O notation:

Operation	Running time
<code>push_back</code>	
<code>pop_back</code>	
<code>at</code>	
<code>size</code>	
<code>empty</code>	

Motivation

This function doesn't know (or care) what kind of container it's printing, or even how it's being printed. What a generic function!

```
template <class Container, class Printer>
void print_container(const Container & things, Printer & printer)
{
    size_t idx = 0;
    while(idx < things.size())
    {
        printer(things.at(idx));
        ++idx;
    }
}
```

All it needs is that the container has a `size` and `at` function, and that the `Printer` class can be used like a function.

Let's see how we call print_container

```
template <class Container, class Printer>
void print_container(const Container & things, Printer & printer)
{
    size_t idx = 0;
    while(idx < things.size())
    {
        printer(things.at(idx)); // this part's still magic...
        ++idx;
    }
}

void main()
{
    Vector<int> vec;
    // fill it up...

    SimplePrint printer;

    print_container(vec, printer);
}
```

A class that looks like a function

In order to make an object act like a function, we *overload operator()*. That's “operator parentheses”.

```
template <class T>
class SimplePrint {
public:
    void operator()(const T & elem) const;
};

void SimplePrint<T>::operator()(const T & elem) const {
    cout << elem << " ";
}
```

A different printer

Here's a printer that only prints certain elements by saving its state (this is why functors are so powerful!).

```
template <class T>
class SillyPrint {
public:
    SillyPrint(): _printed(0) { }
    void operator()(const T & elem);
private:
    size_t _printed;
};

void SillyPrint<T>::operator()(const T & elem) {
    if(++_printed % 2)
        cout << elem << " ";
}
```

Calling print_container with SillyPrint

```
template <class Container, class Printer>
void print_container(const Container & things, Printer & printer)
{
    size_t idx = 0;
    while(idx < things.size())
    {
        printer(things.at(idx)); // this part is not magic!
        ++idx;
    }
}

void main()
{
    List<int> list;
    // fill it up...

    SillyPrint printer;

    print_container(list, silly);
}
```

We accomplished a lot today!

- Write a generic container data structure
- Overload operator parentheses (you should be very familiar with another class that does this...)
- Write functions (or classes) with multiple template arguments
- Write very generic functions to avoid code duplication
- Write a function that applies some operation on each element in a container (this is called “map”)

Disclaimer (not again!)

- The vector class already exists in the STL
- `std::vector` from `#include <vector>`
- Map (and a bunch of other algorithms) also already exist in the STL
- `std::for_each` in `#include <algorithm>`
- Don't reimplement them, make use of them! They are slightly more complicated than ours, but I bet you can figure them out

lecture09: Linked Lists

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

24th June, 2013

Announcements

- mp2 due tonight
- mt1 tomorrow night!
- mt1 review instead of lab tomorrow morning
- mp3 released tonight, mp3.1 extra credit due Friday (6/28)
 - Note that mp3 is a **solo** MP!
 - Please also sign up for a code review slot on Piazza



A neat idea

The `struct` keyword is similar to the `class` keyword. The only difference is the default access scope: `public` for `struct`, `private` for `class`.

Here's an interesting `struct`. What can we do with it?

```
struct Node // same as
{
    Node* next;
};

class Node
{
public:
    Node* next;
};
```

I herd you like classes

We made the `Node` class a `private` member of the `List` class. That way, the `List` class can make use of it without exposing it to the client (encapsulation!).

```
class List
{
    public:
        // ...
    private:
        struct Node
        {
            // what else can go here?
            Node* next;
        };
};
```

A generic container structure

Here's a bare-bones linked list implementation. What do you imagine the public members would be?

```
template <class T>
class List
{
public:
    // ...
private:
    struct Node
    {
        T data;
        Node* next;
    };
    Node* head; // or "first", or "start"
};
```

List ADT Functions

Here are the ADT functions for a List. Compare these to the Vector's.

- 1 insertFront
- 2 insertBack
- 3 removeFront
- 4 removeBack
- 5 size
- 6 empty

Some other useful functions might be

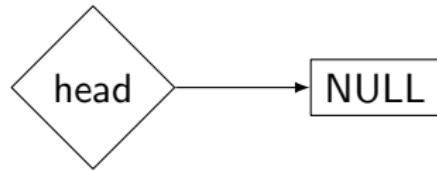
- print
- at
- sort

Warning

- Note that we are implementing the List ADT with a linked list
- We could have chosen to implement the List with an array (in fact, we will look into other options later)
- For now, just be able to distinguish a **List** (an idea, or a concept describing a set of functions) and a **linked list** (an implementation of the List ADT)

Our List's linked list implementation

```
List<int> list;
```



Adding a node

Removing a node

Writing insertFront

```
/** @file list.cpp */

template <class T>
List<T>::List(): head(NULL) { /* nothing */ }

template <class T>
List<T>::Node::Node(const T & newData):
    data(newData), next(NULL) { /* nothing */ }

template <class T>
void List<T>::insertFront(const T & data)
{
    // ok?
    head->next = new Node(data);
}
```

Writing insertFront (take 2)

```
/** @file list.cpp */

template <class T>
void List<T>::insertFront(const T & data)
{
    // now ok?
    if(head == NULL)
    {
        head = new Node(data);
    }
    else
    {
        head->next = new Node(data);
    }
}
```

Writing insertFront (take 3)

```
/** @file list.cpp */

template <class T>
void List<T>::insertFront(const T & data)
{
    // how about now?
    if(head == NULL)
    {
        head = new Node(data);
    }
    else
    {
        Node* front = new Node(data);
        front->next = head;
        head = front;
    }
}
```

insertBack

- How would you write `insertBack`?
 -
- How fast would it be?
 - Any ideas to make it better?
- How about `removeFront` and `removeBack`?
 - Don't want to give too much away about mp3!

Writing the print function iteratively

```
/** @file list.h */

#include <iostream>
using namespace std;

template <class T>
class List
{
public:
    void print() const;

private:
    struct Node
    {
        Node* next;
        T data;
    };
    Node* head;
};
```

```
/** @file list.cpp */

template <class T>
void List<T>::print() const
{
    Node* cur =      ;
    cout << "< ";
    while(          )
    {
        cout << cur->data << " ";
        cur = cur->next;
    }
    cout << ">" << endl;
}

// print out < 1 2 3 4 5 >
```

Writing the print function recursively

```
/**  
 * @file list.cpp  
 * Iterative printing  
 */  
  
template <class T>  
void List<T>::print() const  
{  
    Node* cur = head;  
    cout << "< ";  
    while(cur != NULL)  
    {  
        cout << cur->data << " ";  
        cur = cur->next;  
    }  
    cout << ">" << endl;  
}
```

```
/**  
 * @file list.cpp  
 * Recursive printing  
 */  
template <class T>  
void List<T>::print() const  
{  
    cout << "< ";  
    print(head);  
    cout << ">" << endl;  
}  
  
template <class T>  
void List<T>::print(const Node* cur) const  
{  
}  
}
```

What about a reverse print function?

- Which function would you modify to print out the list backwards?
- Or would you write some entirely different function?
- Hint:

Spicing up linked lists

- A tail pointer (always points to the last element in the list)
- A size variable (update on inserts and removes, initially zero)
- Previous pointers?

lecture10: List Implementations

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

25th June, 2013

Announcements

- No lab today (but you should know that by now!)
- mt1 tonight! Covers all the C++ we've learned
- mp3.1 extra credit due Friday (6/28)



Inserting in the middle of a doubly-linked list

```
template <class T>
void List<T>::insertAfter(size_t num, const T & data)
{
    if(num >= size)                                // make sure we're in range
        return;

                                                // iterate to the position
                                                // we want to insert into

                                                // create new node

                                                // fix pointers

}
```

Three choices for the List ADT

- 1 Array
- 2 Singly-linked list (next pointers only)
- 3 Doubly-linked lists (next and prev)

List running times: linked lists vs arrays

Function	Array	SLL	DLL
insertFront			
insertBack			
removeFront			
removeBack			
insertAtGiven			
removeAtGiven			
insertAtArbitrary			
removeAtArbitrary			

insert at the front and back

remove at the front and back

insert and remove at given locations

A given location is a cell index or pointer to a Node.

insert and remove at arbitrary locations

An arbitrary location is the k^{th} element in the collection. For arrays, this is an index; for linked lists, this is k elements past the head.

lecture11: Stacks

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

26th June, 2013

Announcements

- mp3.1 extra credit due Friday night (6/28)
 - lab_quacks due Saturday night (6/29)

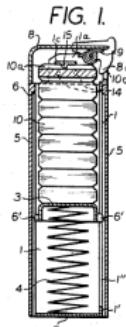


FIG. I

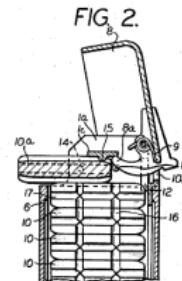


FIG. 2

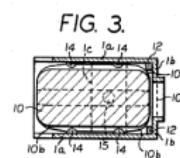
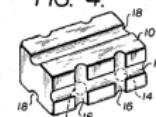


FIG. 3



18 19

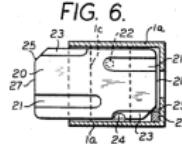
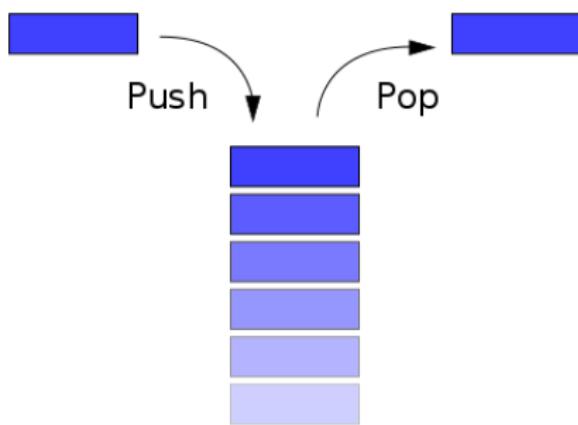


FIG. 6

INVENTOR
EDUARD HAAS
BY *trust of attorney*
ATTORNEY.

The Stack ADT



A Stack is a First-In, Last-Out (FILO) or Last-In, First Out (LIFO) structure. An analogy is a “stack” of papers or cards.

ADT functions include:

- 1 push
- 2 pop
- 3 peek
- 4 size
- 5 empty

What use is a Stack?

- Memory allocation
 - What you've been using for *stack* memory allocations
 - Confusingly, *heap* allocations do not use the heap data structure!
- Doing *traversals*
 - mp4, lab_graphs, possibly mp7
- Parsing
 - CS 421 (compilers), CS 498jhm (natural language processing)
- Undo histories
- Converting a number to binary (how?)
- Balancing parentheses (how?)

Printing a number in binary

```
void printBinary(int n)
{
    Stack<bool> s;
    while(n > 0)
    {
        s.push(n % 2);
        n /= 2;
    }
    while(!s.empty())
        cout << s.pop();
    cout << endl;
}
```

Balancing parentheses with a stack

In order to prepare you for theory classes, this problem is left as an exercise to the student.

What choices do we have?

```
/** @file stack.h */

template <class T>
class Stack
{
public:
    Stack(); // + big 3
    void push(const T & elem);
    T pop();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    // what should go in here?
};
```

Linked list or array?

```
/** @file stack.h */

template <class T>
class Stack
{
public:
    Stack(); // + big 3
    void push(const T & elem);
    T pop();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    size_t _size;
    size_t _capacity;
    T* _data;
};
```

```
/** @file stack.h */

template <class T>
class Stack
{
public:
    Stack(); // + big 3
    void push(const T & elem);
    T pop();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    // Node class...
    Node* _head;
    size_t _size;
};
```

Wait, we made these already...

```
/** @file stack.h */
#include "vector.h"

template <class T>
class Stack
{
public:
    Stack(); // + big 3
    void push(const T & elem);
    T pop();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    Vector<T> _array;
};
```

```
/** @file stack.h */
#include "list.h"

template <class T>
class Stack
{
public:
    Stack(); // + big 3
    void push(const T & elem);
    T pop();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    List<T> _list;
};
```

The array implementation

```
/** @file stack.h */
#include "vector.h"

template <class T>
class Stack
{
public:
    Stack(); // + big 3
    void push(const T & elem);
    T pop();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    Vector<T> _array;
};
```

```
/** @file stack.cpp */

template <class T>
Stack<T>::Stack(): _array(Vector<T>());

template <class T>
void Stack<T>::push(const T & elem)
{
    _array.push_back(elem);
}

template <class T>
T Stack<T>::pop()
{
    return _array.pop_back();
}
```

Decisions for the linked list implementation

```
/** @file stack.h */
#include "list.h"

template <class T>
class Stack
{
public:
    Stack(); // + big 3
    void push(const T & elem);
    T pop();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    List<T> _list;
};


```

```
/** @file stack.cpp */

template <class T>
Stack<T>::Stack(): _list(List<T>());

template <class T>
void Stack<T>::push(const T & elem)
{
    _list.insertFront(elem);
    // or...?
    _list.insertBack(elem);
}

template <class T>
T Stack<T>::pop()
{
    return _list.removeFront();
    // or...?
    return _list.removeBack();
}
```

Linked list intricacies

- When implementing a Stack with a linked list, does it matter where the “top” of the Stack is (head or tail)?
- Does it matter if the linked list is a SLL or a DLL?
- We are concerned with an efficient running time!

Another running time showdown!

Function	Array	SLL (tail is top)	DLL (tail is top)
push			
pop			
peek			

Another running time showdown!

Empirically...

Which implementation do you think is better *empirically*? That is, in practice on actual hardware. We always need to consider real-life performance as well as theoretical performance!

By the way, you should explore the running times on your own when the top of the Stack is the head of the SLL and DLL. What changes?

lecture12: Queues

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

27th June, 2013

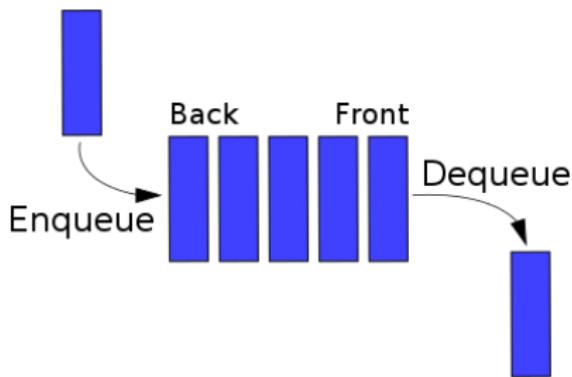
Announcements

- mp3.1 extra credit due tomorrow
- lab_quacks due Saturday night (6/29)
- mp3 due Monday (7/1)



Source: http://i.dailymail.co.uk/i/pix/2010/08/04/article-1300471-0AB0A1D2000005DC-451_468x286.jpg

The Queue ADT



A Queue is a First-In, First-Out (FIFO) or Last-In, Last Out (LILO) structure. An analogy is a line of people waiting for an event.

ADT functions include:

- 1 enqueue
- 2 dequeue
- 3 peek
- 4 size
- 5 empty

What use is a Queue?

- CPU and disk scheduling
 - CS 241 (systems)
- Process communication (as a buffer)
 - CS 241 (systems)
- Doing *traversals* (just like Stacks!)
 - mp4, lab_graphs, possibly mp7
- Parsing
 - CS 421 (compilers), CS 498jhm (natural language processing)
- Letting people into rides at an amusement park (how?)

What choices do we have?

```
/** @file queue.h */

template <class T>
class Queue
{
public:
    Queue(); // + big 3
    void enqueue(const T & elem);
    T dequeue();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    // what should go in here?
};
```

Linked list or array?

```
/** @file queue.h */

template <class T>
class Queue
{
public:
    Queue(); // + big 3
    void enqueue(const T & elem);
    T dequeue();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    size_t _size;
    size_t _capacity;
    T* _data;
};
```

```
/** @file queue.h */

template <class T>
class Queue
{
public:
    Queue(); // + big 3
    void enqueue(const T & elem);
    T dequeue();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    // Node class...
    Node* head;
    size_t _size;
};
```

Wait, we made these already...

```
/** @file queue.h */
#include "vector.h"

template <class T>
class Queue
{
public:
    Queue(); // + big 3
    void enqueue(const T & elem);
    T dequeue();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    Vector<T> _array;
};
```

```
/** @file queue.h */
#include "list.h"

template <class T>
class Queue
{
public:
    Queue(); // + big 3
    void enqueue(const T & elem);
    T dequeue();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    List<T> _list;
};
```

Does any of this seem familiar yet?

Is anyone having *déjà vu*?

Stacks and Queues are opposites: FIFO/LIFO *vs* FILO/LIFO. They are each other's archnemesis.

The array implementation

```
/** @file queue.h */
#include "vector.h"

template <class T>
class Queue
{
public:
    Queue(); // + big 3
    void enqueue(const T & elem);
    T dequeue();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    Vector<T> _array;
};


```

```
/** @file stack.cpp */

template <class T>
Queue<T>::Queue(): _array(Vector<T>());

template <class T>
void Queue<T>::enqueue(const T & elem)
{
}

template <class T>
T Queue<T>::dequeue()
{
}
```

The linked list implementation

```
/** @file queue.h */
#include "list.h"

template <class T>
class Queue
{
public:
    Queue(); // + big 3
    void enqueue(const T & elem);
    T dequeue();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    List<T> _list;
};


```

```
/** @file stack.cpp */

template <class T>
Queue<T>::Queue(): _list(List<T>());

template <class T>
void Queue<T>::enqueue(const T & elem)
{
}

template <class T>
T Queue<T>::dequeue()
{
}
```

Linked list intricacies (again!)

- When implementing a Queue with a linked list, does it matter where the “front” of the Queue is (head or tail)?
- Does it matter if the linked list is a SLL or a DLL?
- We are concerned with an efficient running time!

Your favorite

For linked lists, assume the front of the queue is the head of the list and the back of the queue is the tail of the list.

Function	Array	SLL	DLL
enqueue			
dequeue			
peek			

Your favorite

Parting thoughts

- Again, consider the alternative implementation of a Queue with a (singly/doubly)-linked list: the front is the tail and the back is the head
- Can you think of any other implementations for a Queue besides linked lists and arrays?
 - A circularly linked list with only a head pointer
 - Two Stacks (mp4.1!)
- Can you make a Stack with two Queues?

lecture13: Trees

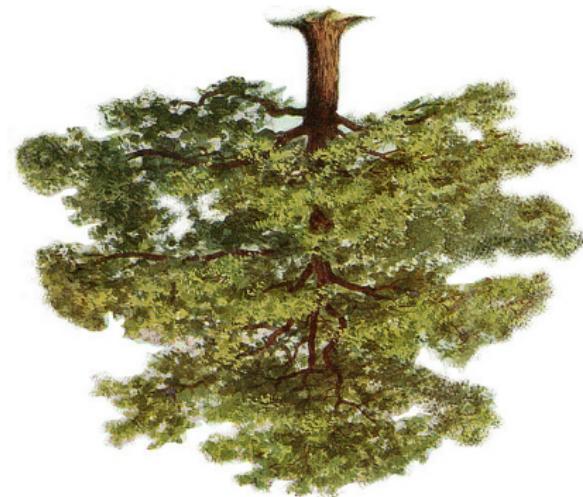
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

1st July, 2013

Announcements

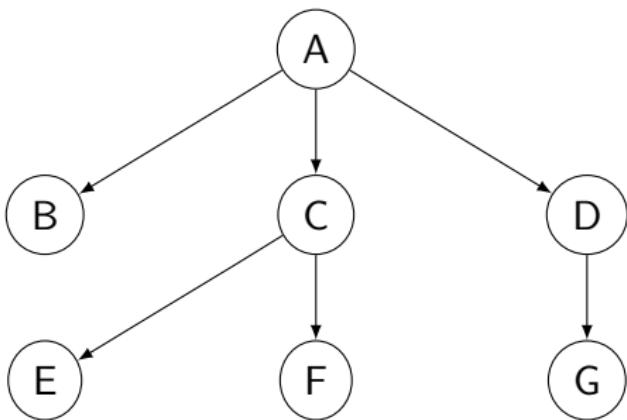
- mp3 due tonight (mp4 released tonight)
- mt1 “solution party” Wednesday (7/3), 10am
- lab_trees due Thursday night (7/4)
- Reminder: no class 7/4
- Friday (7/5) is the last day to drop
- Review on Friday?



More pointers

In linked lists, each Node had one pointer to another Node. What kind of structures can we make if each Node has more than one pointer?

```
template <class T>
struct TreeNode
{
    T data;
    TreeNode* one;
    TreeNode* two;
    TreeNode* three;
    // etc...
};
```

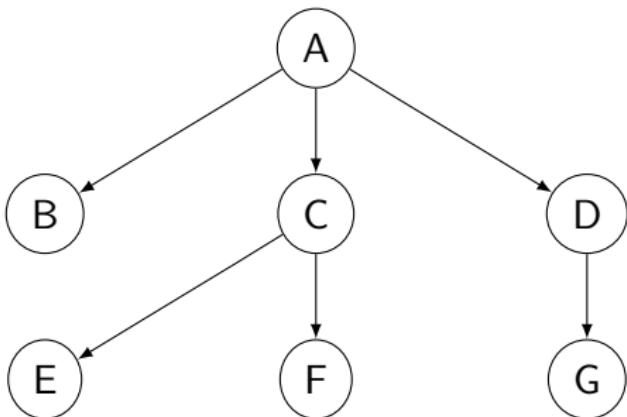


Trees!

We've created a **tree**. Trees are a collection of **vertices** and **edges**. According to Donald Knuth, they are "*...the most important nonlinear structure in computer science.*"

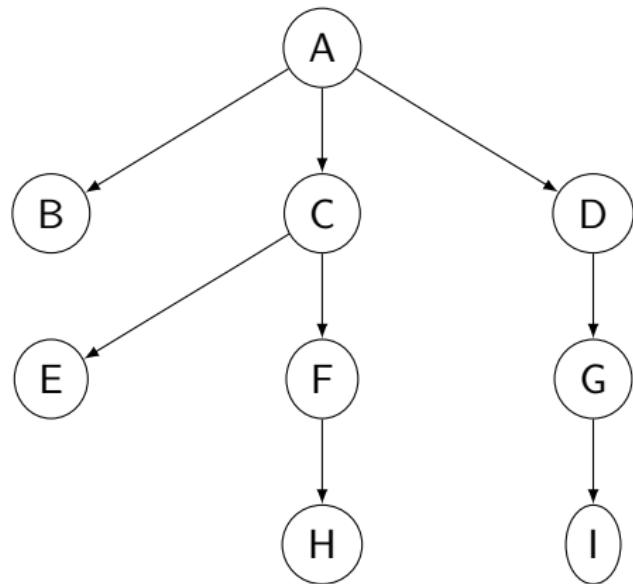
Trees are also...

- 1 Directed
- 2 Acyclic
- 3 Connected

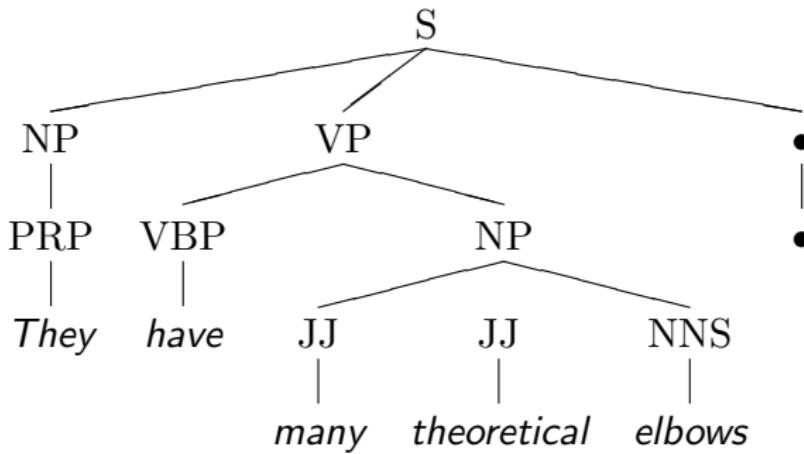


Tree Terminology

- Find an edge that is *not* on the longest path in the tree
- Which node would be called the *root*?
- How many parents does each vertex have?
- Which vertex has the fewest children?
- Which vertex has the most ancestors/descendants?
- List all the nodes in C's subtree
- List all the leaves

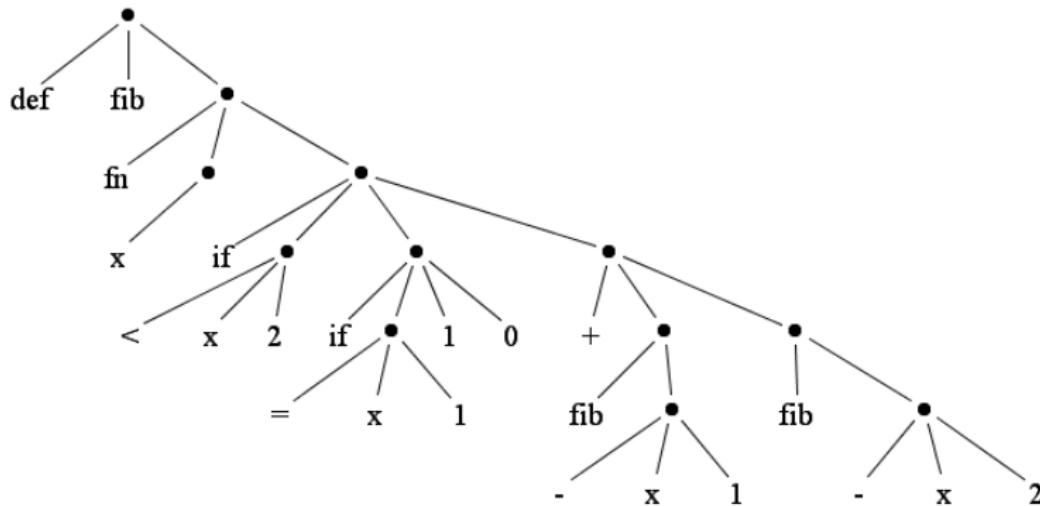


Grammatical Parse Tree



Abstract Syntax Tree

(def fib (fn (x) (if (< x 2) (if (= x 1) 1 0) (+ (fib (- x 1)) (fib (- x 2))))))



Source: http://www.curransoft.com/code/fib_visual_ast_20090414.PNG

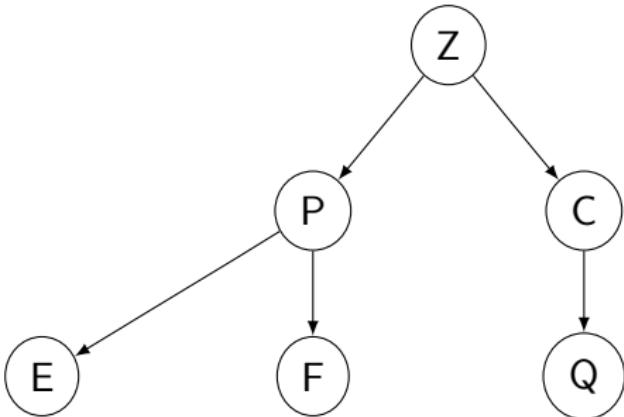
What else?

Recursive definition of a binary tree

A binary tree T is either

- empty, $T = \{\}$
- a root with subtrees,
 $T = \{r, T_{left}, T_{right}\}$,
where $T_{left, right}$ are
binary trees

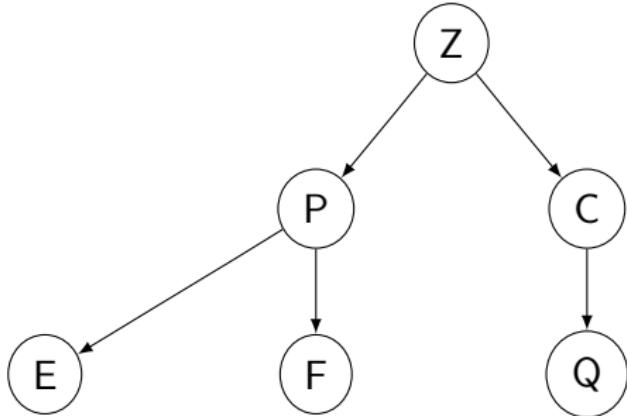
So $T_Z = \{Z, T_P, T_C\}$ and
 $T_C = \{C, T_Q, \{\}\}$



Height of a tree

The height of a tree is the length of the longest path from the root to a leaf.

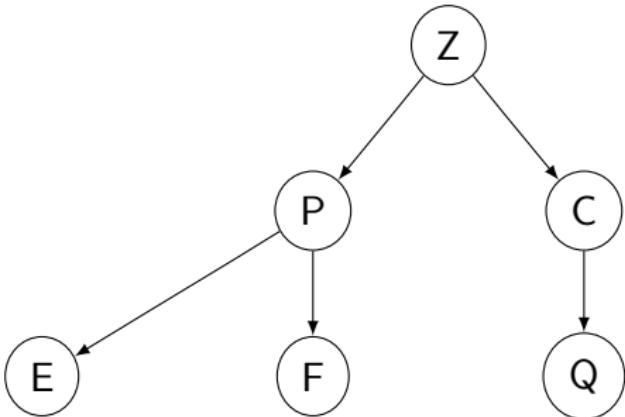
- Can you write a recursive definition of $height(T)$?
- What is the height of the tree to the right?
- What is the height of a tree with only one node?



A full n -ary tree

A full n -ary tree is a tree where each node has either 0 or n children

- Is the tree to the right a *full, binary tree*?
- Can you draw one that is?
- What is the height of the smallest full binary tree?



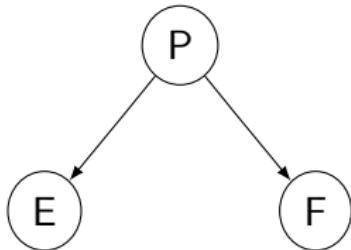
A perfect tree

A perfect tree of height h , or P_h :

- P_{-1} is an empty tree

- if $h > -1$, then

$P_h = \{r, T_{left}, T_{right}\}$, where
 $T_{left,right}$ are P_{h-1}



Is the tree to the left a perfect tree?
What do P_0, P_1 , and P_2 look like?

A *complete tree* has every level full, except possibly the last. Therefore, a perfect tree is a complete tree, but not vice versa.

Quiz!

- What are the minimum number of nodes in a tree of height h ?
- What are the maximum number of nodes in a tree of height h ?
- How many nodes are in a perfect tree of height h ?
- Is every full tree complete?
- Is every complete tree full?

A binary tree implementation

```
/** @file binary_tree.h */

template <class T>
class BinaryTree
{
public:
    BinaryTree(); // + big 3
    // ???
private:
    struct TreeNode
    {
        T data;
        TreeNode* left;      // NULL if no children!
        TreeNode* right;
    };
    TreeNode* root;
};
```

lecture14: Tree Traversals

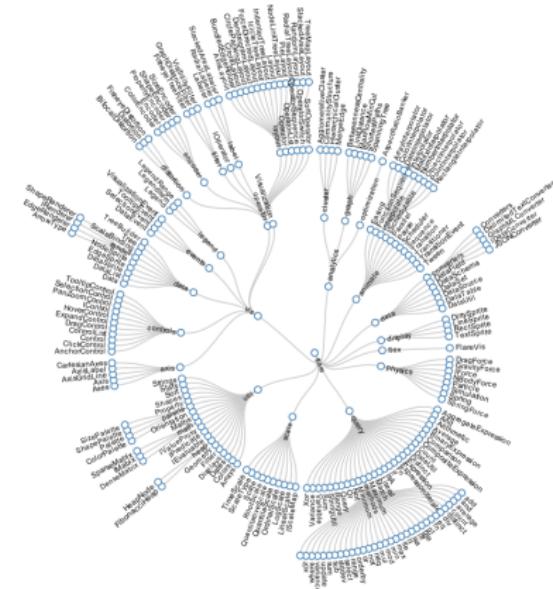
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

2nd July, 2013

Announcements

- lab_trees due **Saturday night** (7/6)
 - Reminder: no class 7/4
 - mp4.1 extra credit due Friday night (7/5)



A binary tree implementation

```
/** @file binary_tree.h */  
  
template <class T>  
class BinaryTree  
{  
public:  
    BinaryTree(); // + big 3  
    // ???  
private:  
    struct TreeNode  
    {  
        T data;  
        TreeNode* left; // NULL if no children!  
        TreeNode* right;  
    };  
    TreeNode* root;  
};
```

How many NULL pointers in a BinaryTree?

Theorem: if there are n items in a binary tree, then there are $n + 1$ NULL pointers. Proof by induction on n .

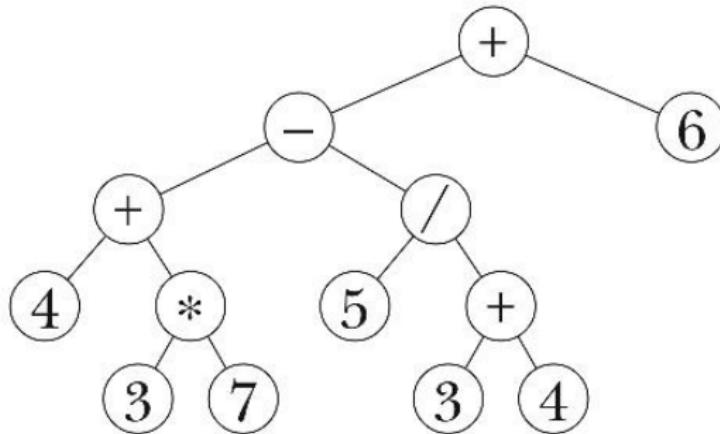
- Case 1: $n = 0 \rightarrow T = \{\}$, which is represented with 1 NULL pointer. $0 + 1 = 1 \quad \checkmark$
- Case 2: $n > 0 \rightarrow T = \{r, T_{left}, T_{right}\}$.
 - Let the size of $T_{left} = a$ and the size of $T_{right} = b$
 - Then $n = a + b + 1$. By the inductive hypothesis, $\forall j < n$, a binary tree with j nodes has $j + 1$ NULL pointers
 - That means we know T_{left} has $a + 1$ and T_{right} has $b + 1$
 - So there are a total of $a + b + 2$ NULL pointers, where $a + b + 1 = n$
 - $a + b + 2 = n + 1 \quad \checkmark$

The BinaryTree Class

```
/** @file binary_tree.h */\n\ntemplate <class T>\nclass BinaryTree\n{\npublic:\n    BinaryTree(); // + big 3\n    void insert(const T & elem);\n    void remove(const T & elem);\n    void traverse() const;\nprivate:\n    struct TreeNode\n    {\n        T data;\n        TreeNode* left;\n        TreeNode* right;\n    };\n    TreeNode* root;\n};
```

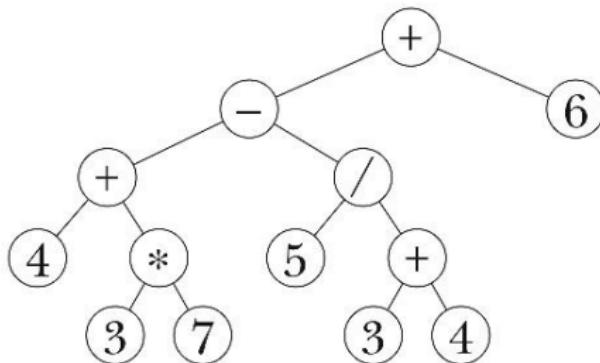
- Let's delay talking about `insert` and `remove` until we talk about the Dictionary ADT
- Today we'll investigate different types of tree traversals and their applications

How should we visit each node?



source: <http://www.eecs.berkeley.edu/bh/ss-pics/parse0.jpg>

How should we visit each node?



We will discuss four kinds of traversals:

- 1 Pre-order
- 2 In-order
- 3 Post-order
- 4 Level order

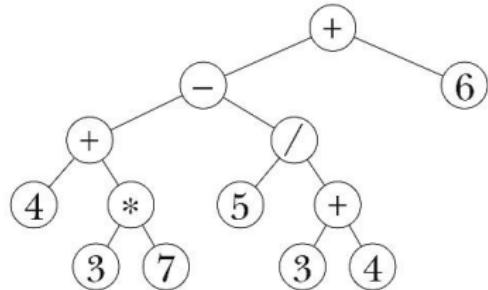
A scheme for visiting each node

- At each node, there are two choices for direction
- After both subtrees are complete, move back up the tree
- Each node is visited three times during the traversal
- For the first three schemes, each visit time corresponds to a different scheme
- The last traversal (level order) uses an *ordering structure* to keep track of the nodes

A pre-order traversal

```
// public interface
template <class T>
void BinaryTree<T>::traverse() const
{
    traverse(root);
}

// private helper
template <class T>
void BinaryTree<T>::
    traverse(const TreeNode* cur) const
{
    if(cur != NULL)
    {
        yell(cur->data);
        traverse(cur->left);
        traverse(cur->right);
    }
}
```



Prints:

An in-order traversal

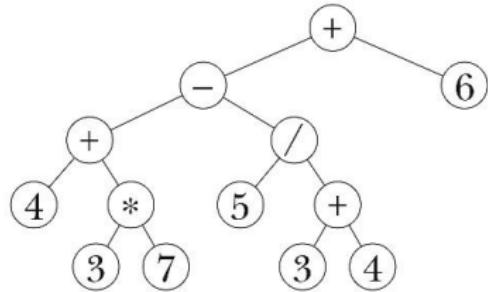
How would you write an in-order traversal?

What should it print?

An in-order traversal

```
// public interface
template <class T>
void BinaryTree<T>::traverse() const
{
    traverse(root);
}

// private helper
template <class T>
void BinaryTree<T>::
    traverse(const TreeNode* cur) const
{
    if(cur != NULL)
    {
        ...
    }
}
```



Prints:

A post-order traversal

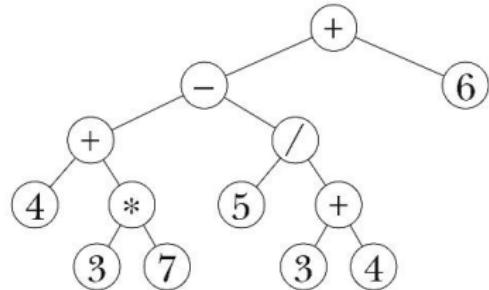
How would you write a post-order traversal?

What should it print?

A post-order traversal

```
// public interface
template <class T>
void BinaryTree<T>::traverse() const
{
    traverse(root);
}

// private helper
template <class T>
void BinaryTree<T>::
    traverse(const TreeNode* cur) const
{
    if(cur != NULL)
    {
        ...
    }
}
```



Prints:

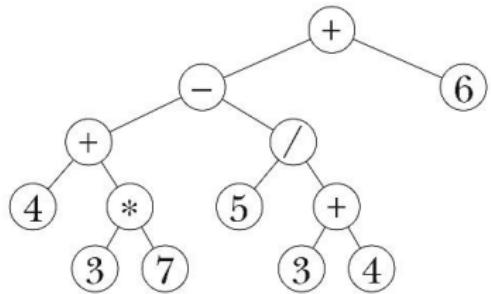
And now for something completely different

What do you think a level order traversal is?

How would you write it?

A level order traversal

```
template <class T>
void BinaryTree<T>::traverse() const
{
    Queue<TreeNode*> q;
    q.enqueue(root);
    while(!q.empty())
    {
        TreeNode* cur = q.dequeue();
        if(cur != NULL)
        {
        }
    }
}
```



Prints:

A few questions

- What is the running time of the traversal functions? Are they all the same?
- Explain the reasoning behind having public and private versions of the first three traversal functions
- What if we wanted to apply *any* function at each node?
- Imagine you are writing the `clear` function, and `delete` is your operation. Which traversal should you use?

lecture15: Binary Search Trees

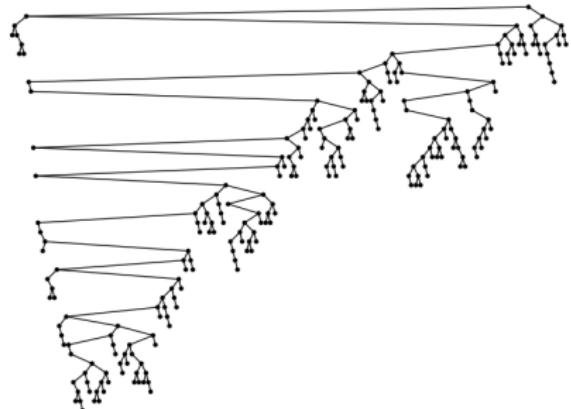
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

3rd July, 2013

Announcements

- mp4.1 extra credit due Friday night (7/5)
- lab_trees due Saturday night (7/6)
- mp4 due Monday night (7/8)
- No class tomorrow! You can come, but I won't be here



The Dictionary ADT

Suppose we had the following data...

Course	Name
125	Lawrence
173	Margaret
225	Cinda
233	Craig
241	Wade
242	Mike

...and we want to be able to retrieve a name given a course

What about...

- UIN → advising record
- student → grade
- RGBAPixel → count
- vertex → incident edges
- URL → web page
- flight number → arrival info

ADT operations

```
/** @file dictionary.h */

template <class K, class V>
class Dictionary
{
public:
    Dictionary(); // + big 3
    void insert(const K & key, const V & value);
    void remove(const K & key);
    V find(const K & key) const;
    size_t size() const;
    bool empty() const;

private:
    // ???
};
```

Our (current) options

1 Vector<Key, Value>

- insert: push_back
- remove: find + removeAt

2 List<Key, Value>

- insert: insertBack
- remove: find + removeAt

3 BinaryTreeNode<Key, Value>

- insert: insert (assume these work as expected)
- remove: find + remove

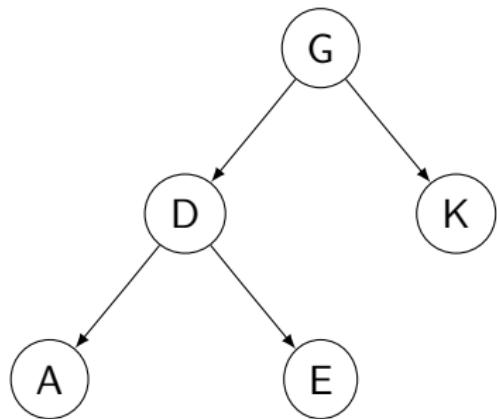
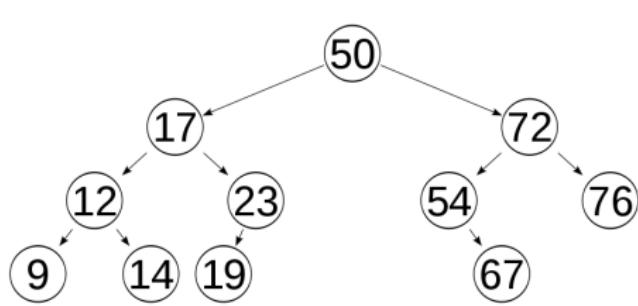
Each node or element in the structure contains a templated key, value pair. What would the running times of each function be?

Performance comparison

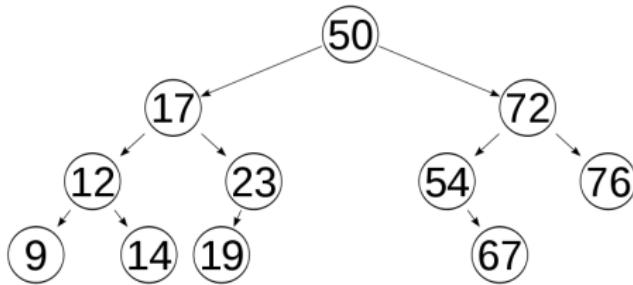
Structure	Insert	Find	Remove
unsorted array			
sorted array			
linked list			
binary tree			

Then what use is a binary tree??

A special kind of binary tree



A special kind of binary tree



- How would you execute the call `find(19)` starting at the root?
- How about `find(58)`?
- Where would 58 go if you had to insert it somewhere?

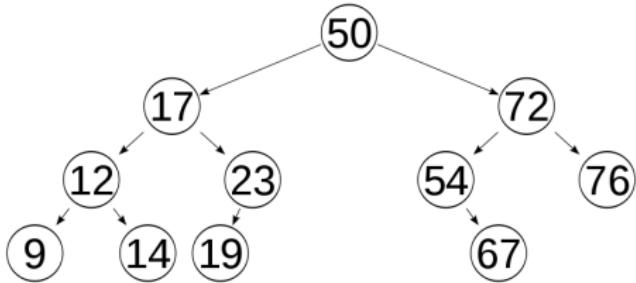
Binary Search Trees

A binary search tree (BST) is a binary tree T , where either

- 1 $T = \{\}$

or

- 1 $T = \{r, T_{left}, T_{right}\}$
- 2 $\forall t \in T_{left}, \text{key}(t) \leq \text{key}(r)$
- 3 $\forall t' \in T_{right}, \text{key}(t') > \text{key}(r)$
- 4 $T_{left, right}$ are BSTs



Our BST class

```
/** @file bst.h */

template <class K, class V>
class BST
{
public:
    BST(); // + big 3
    void insert(const K & key, const V & value);
    V find(const K & key) const;
    void remove(const K & key);

private:
    struct TreeNode
    {
        K key;
        V value;
        TreeNode* left;
        TreeNode* right;
    };
    TreeNode* root;
};
```

*// this class header isn't
// all that different from
// the BinaryTree one. The
// difference is in how the
// functions operate!*

Writing find

```
template <class K, class V>
V BST<K, V>::find(const K & key) const
{
    find(key, root);
}

template <class K, class V>
V BST<K, V>::find(const K & key, const TreeNode* cur) const
{
    if(cur == NULL)

        else if(cur->key == key)

        else if(cur->key < key)

    else

}
```

Writing insert

```
template <class K, class V>
void BST<K, V>::insert(const K & key, const V & value)
{
    insert(key, value, root);
}

template <class K, class V>
void BST<K, V>::insert(const K & key, const V & value, TreeNode* & cur)
{
    if(cur == NULL)

        else if(cur->key < key)

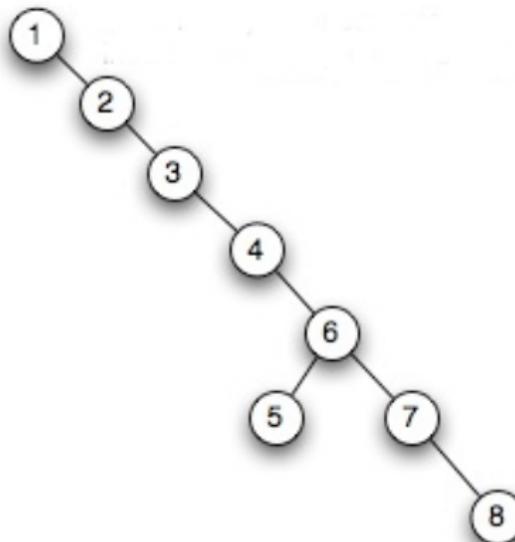
        else

    }
}
```

Better running times?

Structure	Insert	Find	Remove
unsorted array	$O(1)$	$O(n)$	$O(n) + O(n)$
sorted array	$O(\log n) + O(n)$	$O(\log n)$	$O(\log n) + O(n)$
linked list	$O(1)$	$O(n)$	$O(n) + O(1)$
binary tree	$O(n)$	$O(n)$	$O(n) + O(1)$
BST	?	?	—

What about this BST?



lecture16: BST Remove

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

8th July, 2013

ANNOUNCEMENTS

- MP4 DUE TONIGHT
- OPTIONAL REVIEW
INSTEAD OF LAB
TOMORROW MORNING
- MT2 TOMORROW NIGHT



Our BST class

```
/** @file bst.h */

template <class K, class V>
class BST
{
public:
    BST(); // + big 3
    void insert(const K & key, const V & value);
    V find(const K & key) const;
    void remove(const K & key);

private:
    struct TreeNode
    {
        K key;
        V value;
        TreeNode* left;
        TreeNode* right;
    };
    TreeNode* root;
};

};
```

Writing find

```
template <class K, class V>
V BST<K, V>::find(const K & key) const
{
    find(key, root);
}

template <class K, class V>
V BST<K, V>::find(const K & key, const TreeNode* cur) const
{
    if(cur == NULL)

        else if(cur->key == key)

        else if(cur->key < key)

    else

}
```

Writing insert

```
template <class K, class V>
void BST<K, V>::insert(const K & key, const V & value)
{
    insert(key, value, root);
}

template <class K, class V>
void BST<K, V>::insert(const K & key, const V & value, TreeNode* & cur)
{
    if(cur == NULL)

        else if(cur->key < key)

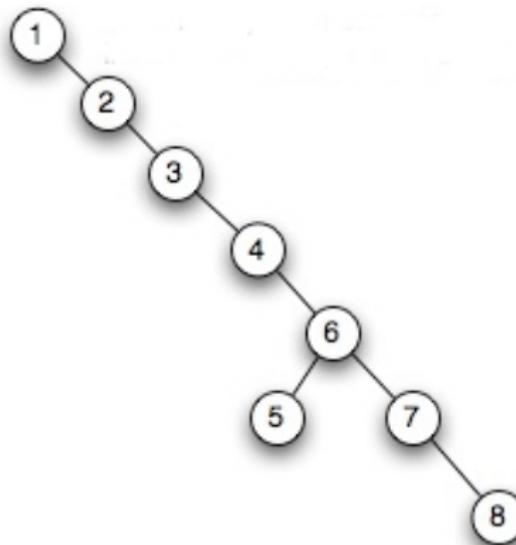
        else

    }
}
```

Better running times?

Structure	Insert	Find	Remove
unsorted array	$O(1)$	$O(n)$	$O(n) + O(n)$
sorted array	$O(\log n) + O(n)$	$O(\log n)$	$O(\log n) + O(n)$
linked list	$O(1)$	$O(n)$	$O(n) + O(1)$
binary tree	$O(n)$	$O(n)$	$O(n) + O(1)$
BST	?	?	—

What about this BST?



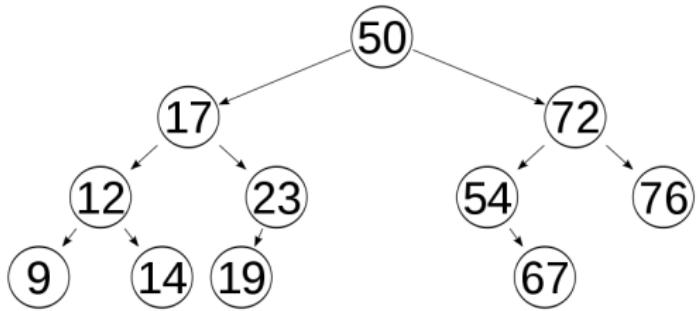
The remove function

```
template <class K, class V>
void BST<K, V>::remove(const K & key)
{
    remove(key, root);
}

template <class K, class V>
void BST<K, V>::remove(const K & key, TreeNode* & cur)
{
    if(cur == NULL)
        return;
    else if(cur->key == key)
        doRemoval(cur);
    else if(cur->key < key)           // This is basically just
        remove(cur->right);           // find, except instead of
    else                                // returning the node we're
        remove(cur->left);           // looking for, we remove it!
}
```

How would you write doRemoval?

- How would you remove 23?
- 72?
- 50?



The remove function

```
// let's break it down based on how many children we have
// did you notice we're passing cur by reference? why?

template <class K, class V>
void BST<K, V>::doRemoval(TreeNode* & cur)
{
    if(cur->left == NULL && cur->right == NULL)
        noChildRemove(cur);
    else if(cur->left != NULL && cur->right != NULL)
        twoChildRemove(cur);
    else
        oneChildRemove(cur);
}
```

Writing noChildRemove

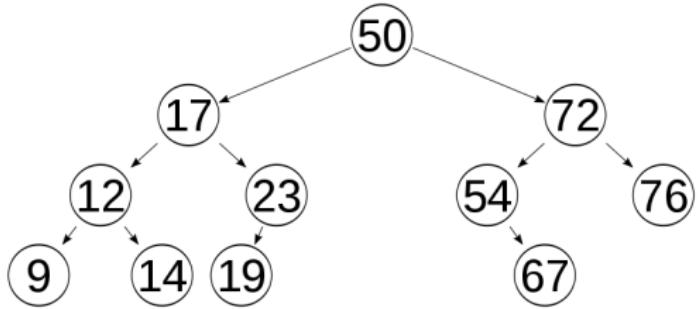
```
// let's do noChildRemove first, it should be the easiest

template <class K, class V>
void BST<K, V>::noChildRemove(TreeNode* & cur)
{
}

}
```

noChildRemove

- How would you remove 9?
- 19?
- 76?



Writing oneChildRemove

```
// now let's do oneChildRemove

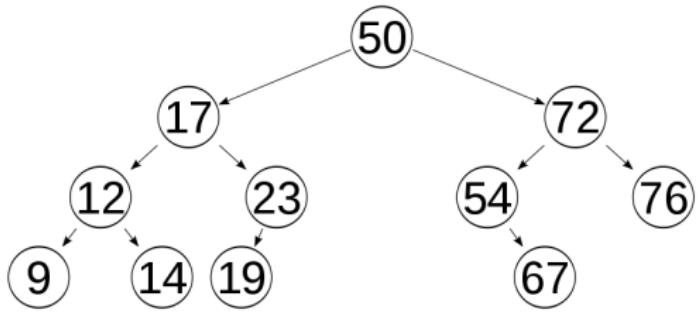
template <class K, class V>
void BST<K, V>::oneChildRemove(TreeNode* & cur)
{
    Node* temp = cur;
    if(cur->left == NULL)

        else

        delete temp;
}
```

oneChildRemove

- How would you remove 23?
- 54?



Writing twoChildRemove

```
template <class K, class V>
void BST<K, V>::twoChildRemove(TreeNode* & cur)
{
    TreeNode* & iop = IOP(cur);

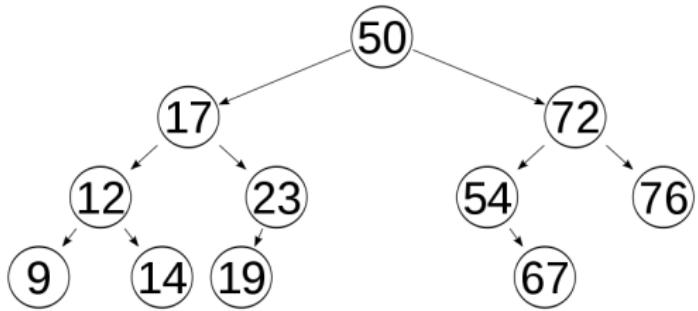
}

// what in the world is this?

template <class K, class V>
typename BST<K, V>::TreeNode* & BST<K,V>::IOP(TreeNode* & cur)
{
    return rightMostChild(cur->left);
}
```

twoChildRemove

- How would you remove 17?
- 72?
- 50?



Oh no...

If only there were some way to ensure that the tree is *balanced*...

Structure	Insert	Find	Remove
unsorted array	$O(1)$	$O(n)$	$O(n)$
sorted array	$O(n)$	$O(\log n)$	$O(n)$
linked list	$O(1)$	$O(n)$	$O(n)$
binary tree	$O(n)$	$O(n)$	$O(n)$
BST	$O(n)$	$O(n)$	$O(n)$
balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$

lecture17: AVL Trees

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

9th July, 2013

Announcements

- mt2 tonight!
- mp5.1 extra credit due Friday (7/12)



An interesting tree

- Can you make a BST that looks like a zig zag?
- What is the insertion order?

Algorithms and Analysis

- BST algorithms depend on the height of the tree; the analysis should be in terms of the amount of data (n) in the tree
- Therefore, we need a relationship between the height, h , and n
- Reminder: $height(T)$ is:
 - -1 if $T = \{\}$,
 - $1 + \max(height(T_{left}), height(T_{right}))$ otherwise

Question 1

What is the maximum number of nodes in a tree of height h ?

$$M(h) = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

We will prove this is correct with the following recurrence:

$M(h) = 1 + 2M(h - 1)$, $M(0) = 1$. Why?

- Base: $M(0) = 2^{0+1} - 1 = 1$ ✓
- Inductive step: assume $M(h - 1) = 2^h - 1$. Then
$$M(h) = 1 + 2(2^h - 1) = 1 + 2^{h+1} - 2 = 2^{h+1} - 1$$
 ✓

Question II

What is the least possible height of a tree of n nodes?

We know from the previous slide $n \leq 2^{h+1} - 1$. That means:

- $n < 2^{h+1}$
- $\log n < h + 1$
- $\log n - 1 < h$
- So a h is $O(\log n)$

Question III and IV

What is the minimum number of nodes in a tree of height h ?

$$n \geq h + 1$$

What is the greatest possible height of a tree with n nodes?

$$h \leq n - 1$$

Fun facts:

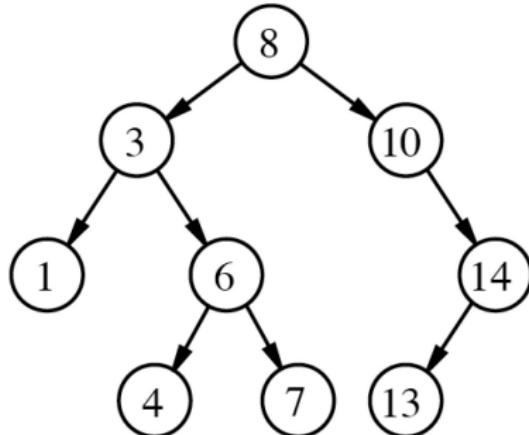
- The height of a BST depends on the order the data is inserted
- Average height is $O(\log n)$, but worst case is $O(n)$

Height balance

- The *height balance* of a tree is defined to be
 $b = \text{height}(T_{\text{left}}) - \text{height}(T_{\text{right}})$
- So, a tree is *height balanced* if:
 - $T = \{\}$, or
 - $T = \{r, T_{\text{left}}, T_{\text{right}}\}$, $b \leq 1$ and $T_{\text{left, right}}$ are height balanced

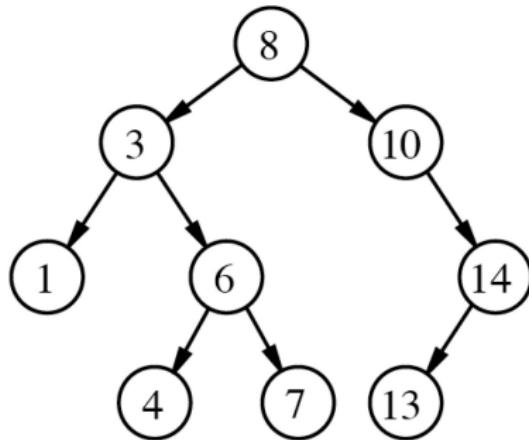
Balance check

- $b = \text{height}(T_{\text{left}}) - \text{height}(T_{\text{right}})$
- A tree is *height balanced* if:
 - $T = \{\}$, or
 - $T = \{r, T_{\text{left}}, T_{\text{right}}\}$, $b \leq 1$ and $T_{\text{left, right}}$ are height balanced
- Is the tree to the right balanced?

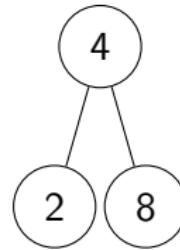
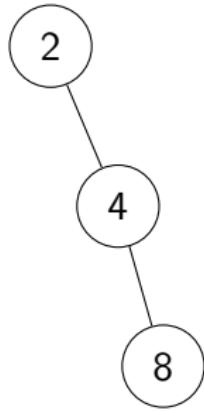


Computing balance at each subtree

- Can you describe a *linear* algorithm to calculate the height at each node?
- Is there an easy way to know a node's height after it is inserted?



Rotating trees: how does the balance change?



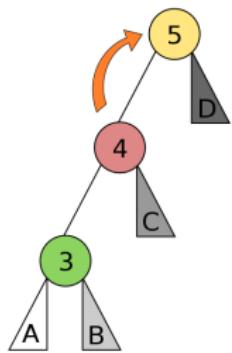
If you had a `TreeNode` pointer to the node containing 2, what code makes this rotation?

Fixing balance with rotations

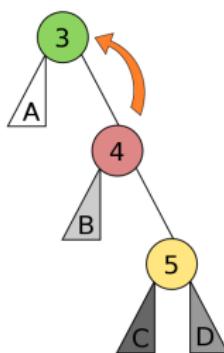
- We just witnessed a *left rotation* about the node containing 2
- There is an analogous *right rotation*
- Left and right rotations turn sticks into mountains
- Using the correct rotations can decrease the height of the tree!
- Note that there could have been other subtrees beneath these nodes

Left and right rotations

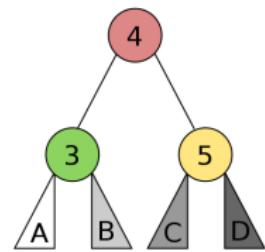
Left Case



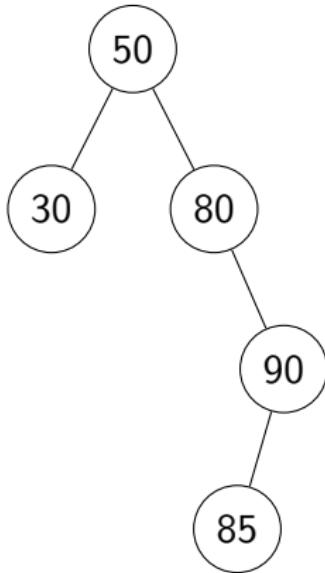
Right Case



Balanced

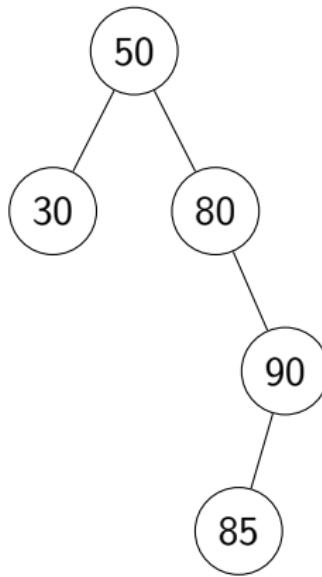


Double rotations

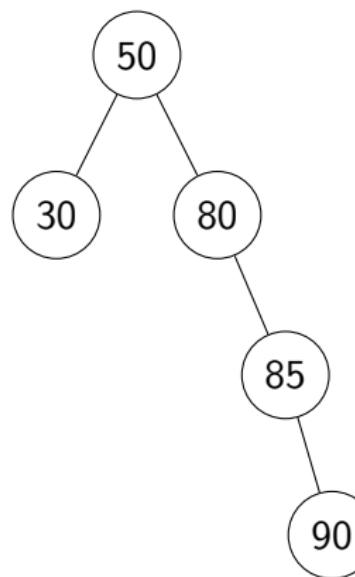


- How can we fix this weird-looking tree?
- Can you make it balanced with a single rotation?
- How about two rotations?

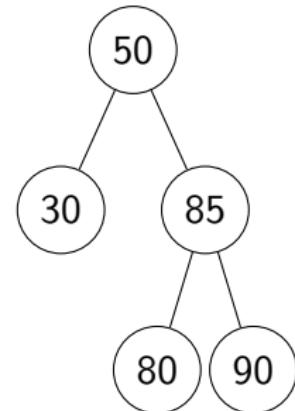
Double rotations



80-85-90:
“boomerang”



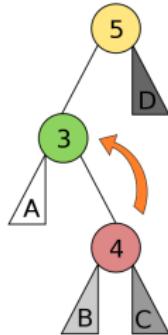
80-85-90:
“stick”



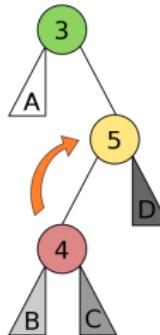
80-85-90:
“mountain”

Left-right and Right-left rotations

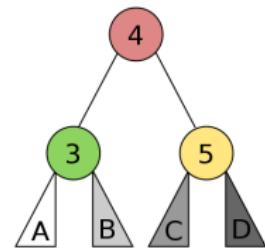
Left Right Case



Right Left Case



Balanced



An efficient Dictionary implementation

- Goal: use rotations to keep a BST balanced
- How does this affect the running time?
- What is the running time of all the rotations?
- How do we know *when* to rotate? Or, how do we know when we are imbalanced?

AVL Trees are the answer!

- AVL trees have a guaranteed $O(\log n)$ height
 - We know from earlier this is the *least possible* height of a tree with n nodes
- They maintain this height with rotations after inserting and deleting
- Heights are stored in each node, making for efficient calculations
 - These stored heights need to be updated after inserting and deleting!

In fact, the AVL tree was the first self-balancing binary search tree to be invented. It is described in the 1962 paper “*An algorithm for the organization of information*” by **Adelson-Velskii** and **Landis**.

lecture18: AVL Analysis

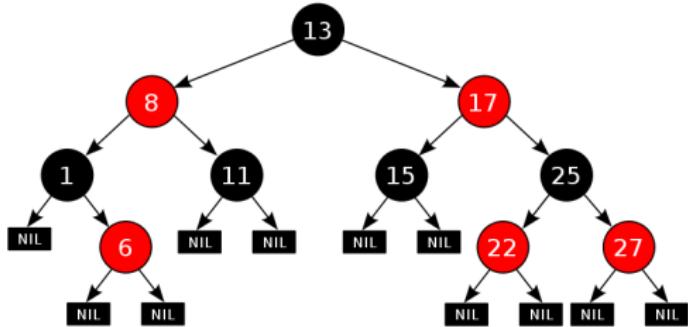
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

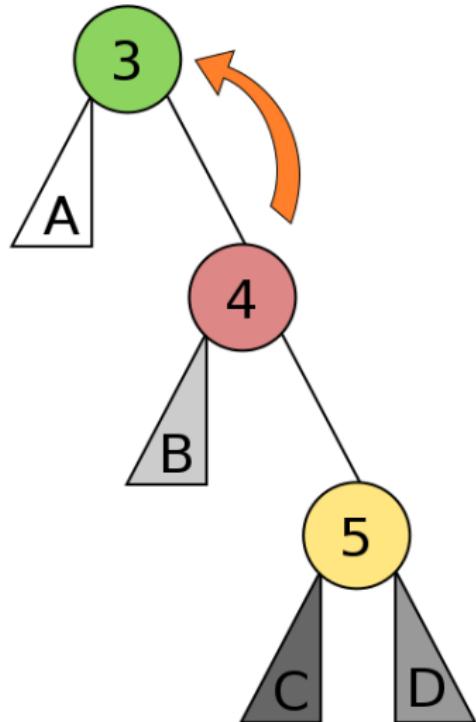
10th July, 2013

Announcements

- lab_huffman
tomorrow, due
Saturday (7/13)
- mp5.1 extra credit
due Friday (7/12)
- mp5 due Monday
(7/15)

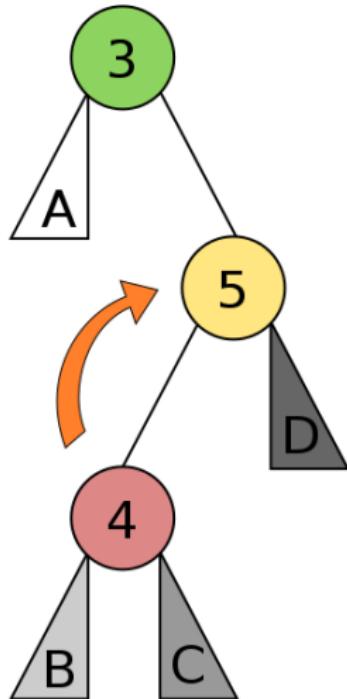


When to rotate left (or right)



- If an imbalance is detected at the subroot containing 3
- ...and if the insertion was in subtrees *C* or *D*
- then a left rotation about 3 rebalances the tree
- (An analogous argument can be made for a right rotation)

When to rotate right-left (or left-right)



- If an imbalance is detected at the subroot containing 3
- ...and if the insertion was in subtrees *B* or *C*
- then a right-left rotation about 3 rebalances the tree
- (An analogous argument can be made for a left-right rotation)

Our AVLTree class

```
/** @file avl.h */

template <class K, class V>
class AVLTree
{
public:
    AVLTree(); // + big 3
    void insert(const K & key, const V & value);
    V find(const K & key);
    void remove(const K & key);

private:
    struct TreeNode
    {
        int height;
        TreeNode* left;
        TreeNode* right;
        K key;
        V value;
    };
};

};
```

Left and right rotations

```
/** @file avl.cpp */

template <class K, class V>
void AVLTree<K, V>::rotateLeft(TreeNode* & t)
{
    // 1. pointer manipulation for rotation
    // 2. update heights (max child height + 1)
}

template <class K, class V>
void AVLTree<K, V>::rotateRight(TreeNode* & t)
{
    // 1. pointer manipulation for rotation
    // 2. update heights (max child height + 1)
}

// you will write these in lab_avl!
```

leftRight and rightLeft rotations

```
/** @file avl.cpp */

template <class K, class V>
void AVLTree<K, V>::rotateLeftRight(TreeNode* & t)
{
    // this function should only be two lines...
}

template <class K, class V>
void AVLTree<K, V>::rotateRightLeft(TreeNode* & t)
{
    // this function should only be two lines...
}
```

Insert (the private helper)

```
template <class K, class V>
void AVLTree<K, V>::insert(TreeNode* & subtree, const K & key, const V & value)
{
    if(subtree == NULL) {
        subtree = new TreeNode(key, value);
    } else if(key < subtree->key) {
        insert(subtree->left, key, value);
        int balanceFactor = // ?
        int leftSideBalanceFactor = // ?
        // detect and perform necessary rotations
    } else {
        insert(subtree->right, key, value);
        int balanceFactor = // ?
        int rightSideBalanceFactor = // ?
        // detect and perform necessary rotations
    }

    // update subtree's height
}
```

Remove

```
template <class K, class V>
void AVLTree<K, V>::remove(TreeNode* & subtree, const K & key)
{
    // same as BST remove, except when traveling back up the tree,
    // we need to re-adjust the TreeNode heights

    // on an insert, at most one rotation is necessary; but here in
    // delete, we can have as many as O(log n)
}
```

Interactive example

Another interactive example

Building the worst-case tree

- Putting an upper bound on the height for a tree of n nodes is the same as putting a lower bound on the number of nodes in a tree of height h
- Define $S(h)$ to be a *sparse* AVL tree; that is, an AVL tree of height h with the minimum number of nodes
- That means $S(-1) = 0, S(0) = 1, S(1) = 2, \dots$
- ...and that $S(h) = 1 + S(h - 1) + S(h - 2)$. Why?

Drawing the worst-case AVL trees

Simplifying and “solving” the recurrence

- We claimed $S(h) = 1 + S(h - 1) + S(h - 2)$
- We know $S(h) \geq S(h - 1) + S(h - 2)$
- Furthermore, we know $S(h) \geq 2 \cdot S(h - 2)$

Can we guess a closed form? $S(h) \geq 2^{\frac{h}{2}}$ makes sense. Why?

- $S(h) \geq 2 \cdot S(h - 2)$
- $S(h) \geq 2 \cdot (2 \cdot S(h - 4))$
- $S(h) \geq 2 \cdot (2 \cdot (2 \cdot S(h - 6)))\dots$
- We see this repeat $\frac{h}{2}$ times since we subtract by 2 each time from h to 0

Proving our guess is correct

Claim: an AVL tree of height h has at least $2^{\frac{h}{2}}$ nodes, $h \geq 0$.

- Case 1: if $h = 0$, then $S(h) = 1 \geq 2^{\frac{0}{2}} = 1$ ✓
- Case 2: if $h = 1$, then $S(h) = 2 \geq 2^{\frac{1}{2}} = 1.414$ ✓
- Case 3: if $h > 1$, then by the IH that says $\forall j < h$, an AVL tree of height j has $\geq 2^{\frac{j}{2}}$ nodes and since $S(h) \geq 2 \cdot S(h-2)$, we know that $S(h) \geq 2 \cdot 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}}$ ✓

That means $n \geq 2^{\frac{h}{2}} \rightarrow \log n \geq \frac{h}{2} \rightarrow h \leq 2 \cdot \log n \dots$

So we know h is $O(\log n)$.

A cute proof

Recall the Fibonacci numbers:

$F(n) = F(n - 1) + F(n - 2)$, $F(0) = 0$, $F(1) = 1$. What happens if we plot the values of the Fibonacci numbers along with $S(h)$?

h	0	1	2	3	4	5	6	7	8	...
$S(h)$	1	2	4	7	12	20	33	54	88	...
$F(h + 3)$	2	3	5	8	13	21	34	55	89	...

Do you see any connection?

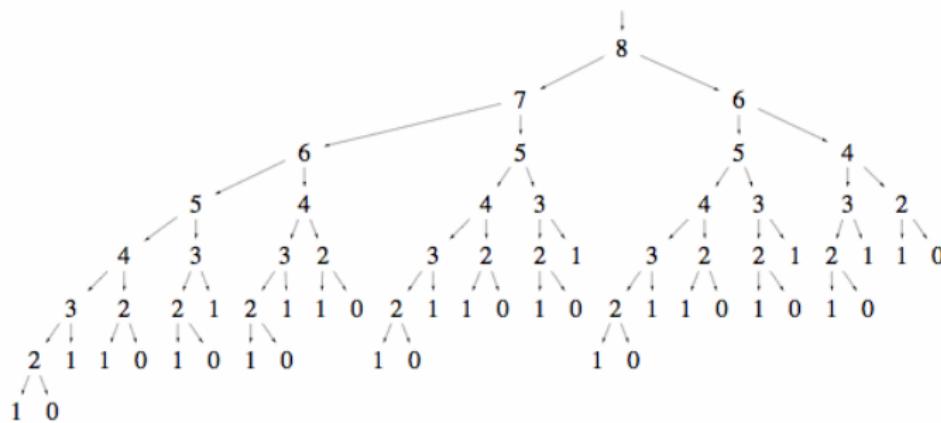
$S(h) = F(h + 3) - 1$, could you prove it?

The Fibonnaci Function

```
size_t fib(size_t n)
{
}
}
```

Fibonacci recursive calls

It looks like we're doing a lot of redundant work!



Can we cache our answers somehow... with a dictionary?

Memoization and dynamic programming

```
#include "avltree.h"

size_t fib(size_t n, AVLTree<size_t, size_t> & cache)
{
    if(n == 0 || n == 1)
        return n;

    if(cache.keyExists(n))
        return cache.find(n);

    size_t answer = fib(n - 1, cache) + fib(n - 2, cache);
    cache.insert(n, answer);

    return answer;
}
```

Memoized Fibonacci using the STL

```
#include <map> // Red-Black tree Dictionary ADT in C++
using namespace std;

size_t fib(int n, map<size_t, size_t> & cache)
{
    if(n == 0 || n == 1)
        return n;

    map<size_t, size_t>::iterator it = cache.find(n);
    if(it != cache.end())
        return it->second;

    size_t answer = fib(n - 1, cache) + fib(n - 2, cache);
    cache[n] = answer;

    return answer;
}
```

lecture19: Fun with Trees

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

11th July, 2013

Announcements

- mp5.1 extra credit due tomorrow
- lab_huffman due Saturday (7/13)
- mp5 due Monday (7/15)



© Gavin Oliver

Sort

```
// when the function exits, the Vector parameter holds the sorted keys
// hint: use insertBack to add elements to the Vector
void sort(const TreeNode* subtree, Vector<K> & sorted)
{
    // ...
}

// what is the running time?
```

AVLCheck

```
// return true if this is an AVL tree. Assume you have a height
// function and that you know the tree is a BST
bool AVLCheck(const TreeNode* subtree)
{
    // what is the running time?
```

nearestKey

```
// given a query key, return the key in the tree that is closest
K nearestKey(const TreeNode* subtree, const K & query, const K & best)
{
    if (subtree == NULL)
        return best;
    if (subtree->key == query)
        return subtree->key;
    if (query < subtree->key)
        best = nearestKey(subtree->left, query, best);
    else
        best = nearestKey(subtree->right, query, best);
    return best;
}

// what is the running time?
```

rangeQuery

```
// prints all keys on the interval [a, b]
void rangeQuery(const TreeNode* subtree, const K & a, const K & b)
{
    // ...
}

// what is the running time?
```

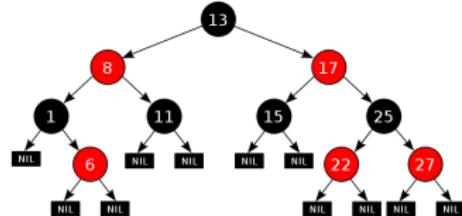
Lowest Common Ancestor

```
// return the lowest node in the tree that could be a
// parent to both a and b
K LCA(const TreeNode* subtree, const K & a, const K & b)
{
    // what is the running time?
```

Red-Black Trees

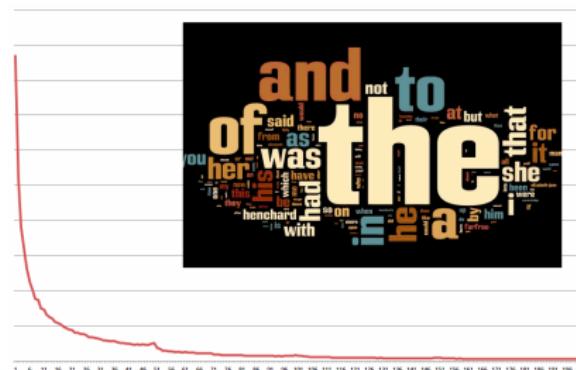
RB trees are less rigidly balanced than AVL trees; this leads to faster insertion times, but slower retrieval.

- Find: $O(\log n)$, Insert: $O(\log n)$,
Delete: $O(\log n)$
- Rotations are performed to keep the
RB properties enforced:
 - 1 A node is either red or black
 - 2 The root is black
 - 3 All leaves are black
 - 4 Both children of every red node are
black
 - 5 Every simple path from a node to its
descendant leaves contains the same
number of black nodes



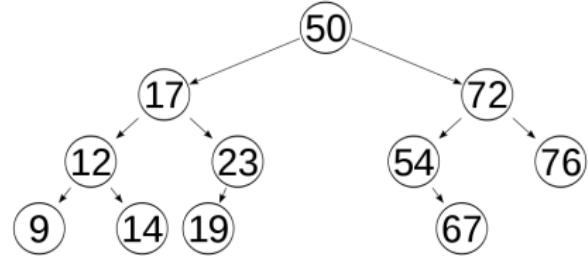
Splay Trees

- Find: $O(\log n)*$, Insert: $O(\log n)*$, Delete: $O(\log n)*$
 - Whenever find or insert is called, rotate that key to the root (how?)
 - What kind of data will benefit from such a structure?



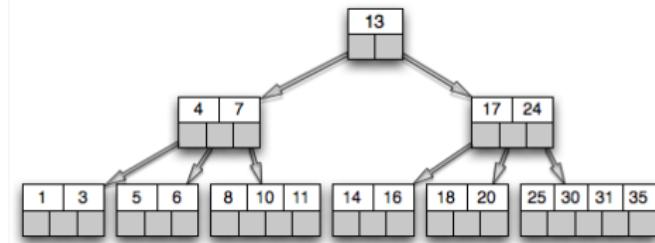
Scapegoat Trees

- Find: $O(\log n)$, Insert: $O(\log n)*$, Delete: $O(\log n)*$
- Ensures that $\forall T$,
 $\text{size}(T_{\text{left}}) \leq \alpha \cdot \text{size}(T)$,
 $\text{size}(T_{\text{right}}) \leq \alpha \cdot \text{size}(T)$,
 $\alpha \in [.5, 1]$
- If this is not satisfied, the entire subtree is rebuilt to be completely balanced in $O(n)$ time



B-Trees

- *Out-of-core* data structure
- $O(\log n)$ running times, but also allows sequential access in nodes
- Designed mainly for use in filesystems or databases (more in CS 411)
- The tree structure is built to minimize the number of expensive disk seeks
- Built by splitting large nodes in half, creating a new parent



Quadtrees and KD-trees

mp5 and mp6!

lecture20: Priority Queue ADT

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

15th July, 2013

Announcements

- mp5 due tonight
- lab_avl tomorrow, due Thursday night (7/18)
- mp6 out tonight, mp6.1 extra credit due Friday night (7/19)



Motivation

The wealthy owner of an amusement park has many connections. His acquaintances (both close and distant) use this to their advantage when they are waiting in line there.

Each patron at the amusement park has some level of familiarity with the rich owner, measured as $f \in [0, 1]$. When new spots are available on the rides, those waiting in line with the highest familiarity get to go on first.

The software used to implement this queueing system makes use of a new ADT: the *PriorityQueue*

The amusement park's queueing system

```
PriorityQueue<Person> line;  
  
Person a("Bob", .34);           // familiarity = .34  
Person b("Sally", .76);         // familiarity = .76  
Person c("Fred", .66);          // familiarity = .66  
  
line.push(a);  
line.push(b);  
line.push(c);  
  
line.pop();                    // returns Sally  
  
Person d("Jim", .50);          // familiarity = .50  
line.push(d);  
  
line.pop();                    // returns Fred  
line.pop();                    // returns Jim  
line.pop();                    // returns Bob
```

The PriorityQueue ADT

```
/** @file priority_queue.h */  
  
template <class T>  
class PriorityQueue  
{  
public:  
    PriorityQueue(); // + big 3  
    void push(const T & elem);  
    T peek() const;  
    T pop();  
    size_t size() const;  
    bool empty() const;  
  
private:  
    // ???  
};
```

Let's assume we can compare type T objects with operator<.

That is, if T == Person, operator< would work with respect to each person's familiarity.

How would you implement a PriorityQueue?

Implementation options:

- 1 Unsorted or sorted array?
- 2 Unsorted or sorted linked list?
- 3 Balanced or unbalanced BST?

Remember, we need to support these PriorityQueue ADT functions:

- 1 push (any element)
- 2 pop (the highest priority element)
- 3 peek (show the next element that will be popped)

PQ: A sorted array

Operation	How?	Running Time
push	search + insert	
pop	search + remove	
peek	search	

PQ: A sorted LL

Operation	How?	Running Time
push	search + insert	
pop	search + remove	
peek	search	

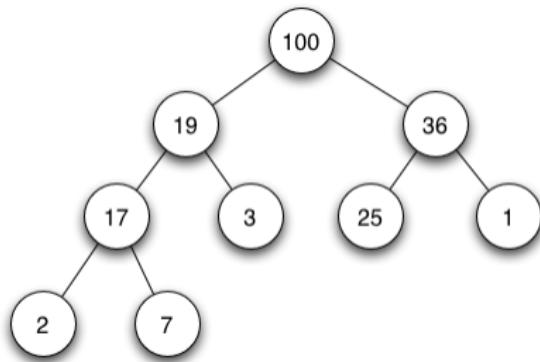
PQ: A balanced BST

Operation	How?	Running Time
push	search + insert	
pop	search + remove	
peek	search	

PQ: A binary heap

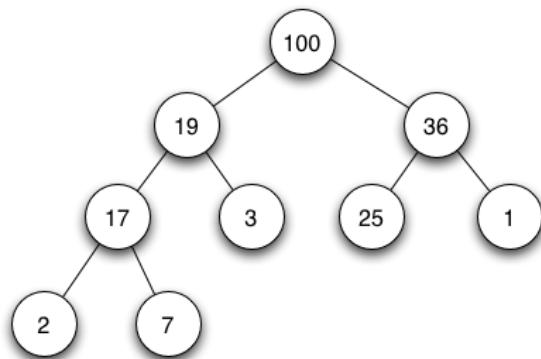
Operation	How?	Running Time
push	?	$O(\log n)$
pop	?	$O(\log n)$
peek	?	$O(1)$

Binary Heaps



- What kind of tree structure is this?
- Can you describe any properties?
- Imagine this is a max-heap; that is, higher numbers have a higher priority

Binary Heaps

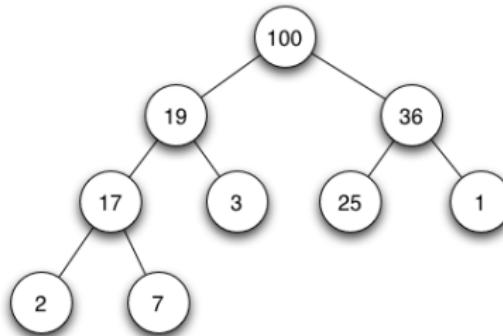


Recursive definition: a heap T is a complete binary tree and

- $T = \{\}$ is a heap
- $T = \{r\}$ is a heap
- $T = \{r, T_{left}, T_{right}\}$ and
 - $T_{left, right}$ are not both empty
 - $key(r) > key(T_{left})$
 - $key(r) > key(T_{right})$
 - $T_{left, right}$ are both heaps

Heap Implementation

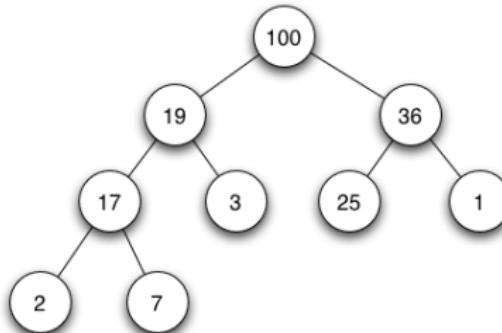
A heap is usually stored as an array, *not* as a pointer-based tree!
We can do this efficiently because we know the heap is a complete tree.
(Why is this important?)



100	19	36	17	3	25	1	2	7	X	X	X	X	X	X
-----	----	----	----	---	----	---	---	---	---	---	---	---	---	---

Heap Implementation

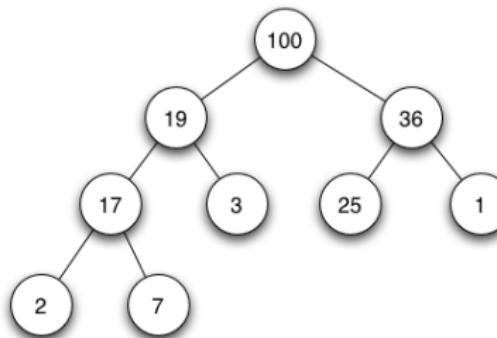
Given the index of a subtree, how can you calculate the indices of its children and parent?



100		19		36		17		3		25		1		2		7		X		X		X		X		X
-----	--	----	--	----	--	----	--	---	--	----	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---

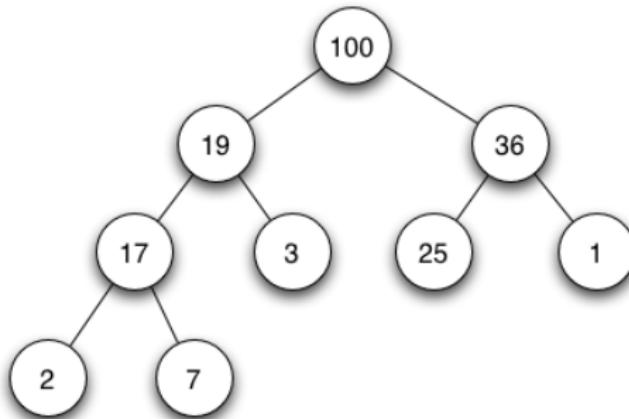
Heap Implementation

Sometimes it makes the calculations easier to have the heap start at index 1:



X	100	19	36	17	3	25	1	2	7	X	X	X	X	X
---	-----	----	----	----	---	----	---	---	---	---	---	---	---	---

Could you write peek? What is the running time?



X	100	19	36	17	3	25	1	2	7	X	X	X	X	X	X
---	-----	----	----	----	---	----	---	---	---	---	---	---	---	---	---

Our heap implementation

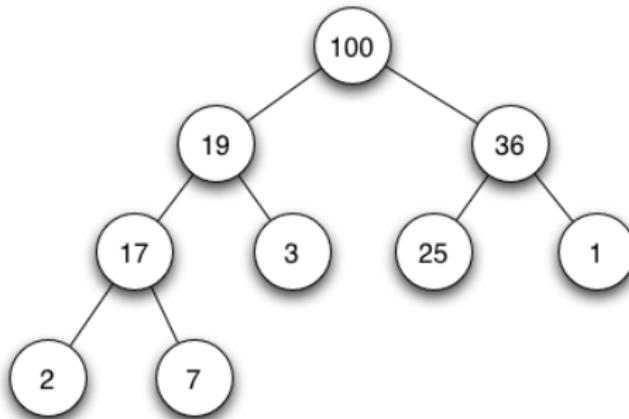
Let's implement a PriorityQueue with a heap, and implement the heap with a Vector (which is a dynamically resizing array).

Let's assume in our heap class that the function `root()` returns 0 or 1 depending on how we set up our Vector to hold our elements.

In that case, `peek` would simply be

```
template <class T>
T Heap<T>::peek() const
{
}
```

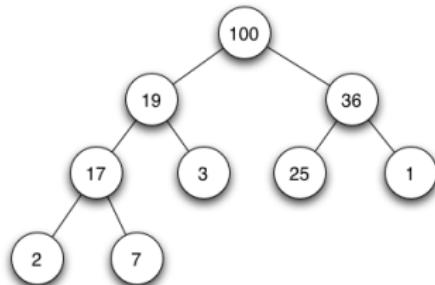
Could you write pop? What is the running time?



X	100	19	36	17	3	25	1	2	7	X	X	X	X	X	X
---	-----	----	----	----	---	----	---	---	---	---	---	---	---	---	---

Writing pop

```
template <class T>
T Heap<T>::pop()
{
```



```
}
```

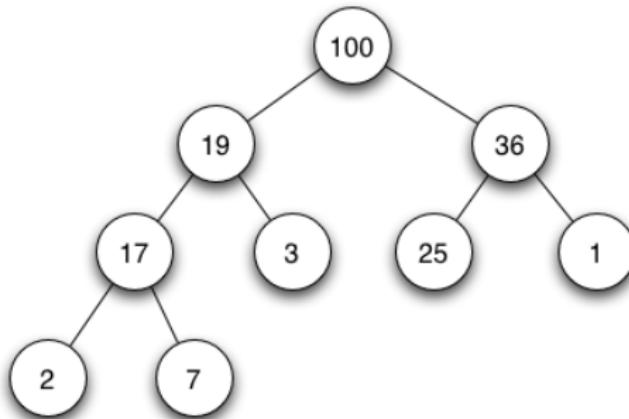
X	100	19	36	17	3	25	1	2	7	X	X	X	X	X	X
---	-----	----	----	----	---	----	---	---	---	---	---	---	---	---	---

HeapifyDown

- While we are not at a leaf...
 - Swap the current value with the value of the higher priority child
 - call HeapifyDown on the child index you swapped with

This sifts elements down into the heap until the heap property is restored. You will write this function in lab_heaps.

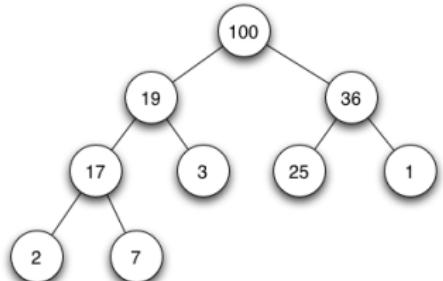
Could you write push? What is the running time?



X	100	19	36	17	3	25	1	2	7	X	X	X	X	X	X
---	-----	----	----	----	---	----	---	---	---	---	---	---	---	---	---

Writing push

```
template <class T>
void Heap<T>::push(const T & elem)
{
}
```



X	100	19	36	17	3	25	1	2	7	X	X	X	X	X	X
---	-----	----	----	----	---	----	---	---	---	---	---	---	---	---	---

HeapifyUp

- While we are not at the root...
 - Swap the current value with the value of the parent if your priority is higher
 - call HeapifyUp on the parent index if you swapped with it

This sifts elements up into the heap until the heap property is restored. You will write this function in `lab_heaps`.

lecture21: Heaps and Sorting

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

16th July, 2013

Announcements

- lab_avl due Thursday night (7/18)
- mp6.1 extra credit due Friday night (7/19)



Making a heap

Say we get an array of random elements. What is the easiest way to turn it into a heap so we can use it as a priority queue?

- Sort it?
- Something else?

Of course, we will have to consider the running time. We don't want to do anything inefficient!

buildHeap by sorting

We know that a sorted array satisfies the heap property. So in order to write buildheap, we'll just call our favorite sorting algorithm!

```
template <class T>
void Heap<T>::buildHeap()
{
    mergeSort(_elems);
    // or...
    quickSort(_elems);
    // or...
    bogoSort(_elems);
}
```

buildHeap with heapifyUp

Can we use functions that already exist in the heap class? We don't need a fully-sorted array to be our heap.

```
template <class T>
void Heap<T>::buildHeap()
{
}
```

This `buildHeap` uses repeated inserts into the heap. Running time: $O(n)$ `heapifyUps`, each taking $O(\log n)$ gives a $O(n \log n)$ running time, the same as mergesort!

But in addition... this is in-place! We don't need to allocate any extra memory.

buildHeap with heapifyDown

Can we make a similar buildHeap with heapifyDown?

```
template <class T>
void Heap<T>::buildHeap()
{
}
```

What is the running time of this? We know that it is proportional to the height of the subtree that heapifyDown is being called on...

buildHeap: heapifyUp vs heapifyDown

The worst case for heapifyDown is when it's called at the root; then, it has to do at worst h operations. The worst case for heapifyUp is when it's called on every single leaf and it has to travel to the top; this is about $\frac{n}{2} \cdot h$ operations.

We already know that using heapifyUp will run in $O(n \log n)$ time. The running time using heapifyDown *is proportional to the sum of the heights of the subtrees it is called on.*

Let's model this as a recurrence: $S(h) = 2 \cdot S(h - 1) + h$, $S(0) = 0$. I claim the solution is $2^{h+1} - h - 2$. What is the running time of buildHeap if this is true?

Proving buildHeap is linear

$$S(h) = 2 \cdot S(h-1) + h, S(0) = 0. \text{ Solution: } 2^{h+1} - h - 2.$$

Proof of solution to recurrence. Consider an arbitrary $h \geq 0$.

- Case 1: $h = 0, S(0) = 2^{0+1} - 0 - 2 = 0. \checkmark$
- Case 2: $h > 0$, by an IH that says $\forall j < h, S(j) = 2^{j+1} - j - 2$, we know $S(h-1) = 2^h - h - 1$. So
$$S(h) = 2 \cdot S(h-1) + h = 2(2^h - h - 1) + h = 2^{h+1} - 2 - h. \checkmark$$

The number of nodes n is proportional to $2^{h+1} - 2 - h$

In terms of number of nodes

$$h \leq \log n$$

$$h + 1 \leq \log n + 1$$

$$2^{h+1} \leq 2^{\log n + 1}$$

$$2^{h+1} - 2 - h \leq 2^{\log n + 1} - 2 - \log n$$

Simplifying,

$$2^{h+1} - 2 - h \leq 2^{\log n + 1} - 2 - \log n = n - 1 - \log n = O(n)$$

Sorting with a heap

How could you implement a sorting algorithm using a heap?

```
template <class T>
Vector<T> heapsort(const Vector<T> & toSort)
{
    Vector<T> sorted;
    Heap<T> h(toSort);    // constructs heap from vector

    while(!h.empty())
        sorted.push_back(h.pop());

    return sorted;
}
```

What if we don't want to allocate extra memory?

Sorting in-place within the heap class

```
// assume we have a valid heap

while(_size > 0)
{
    }

reverse(_elems);          // is this in-place?
_size = elems.size();    // set correct value of _size
```

Runtime analysis of heapSort

- 1 buildHeap: $O(n)$
- 2 heapifyDown n times: $O(n \log n)$
- 3 reverse: $O(n)$

Final running time: $O(n \log n)$.

Why is heapsort interesting?

Algorithm	In-place?	Stable?	Average	Worst
Mergesort	No	Yes		
Quicksort	Yes	No		
Heapsort	Yes	No		

The top k elements

Describe an algorithm that given an unsorted array of length n , returns the top k elements, where $k \ll n$. It must run in time better than $O(n \log n)$.

Use a heap! This can give you either

- $O(n) + O(k \log n)$, or
- $O(n \log k)$

depending on your algorithm.

We can actually do this in linear time if we use an algorithm called quickselect (discussed more during mp6 and CS 473).

lecture22: Hashing

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

17th July, 2013

Announcements

- lab_avl due tomorrow (7/18)
- mp6.1 extra credit due Friday night (7/19)
- lab_heaps out tomorrow, due Saturday night (7/20)



Your task

Say you have a list of names, and you want to figure out which area is associated with each name. Can you use an efficient data structure to do this?

Name	Area
Dan Roth	Machine Learning and NLP
Chengxiang Zhai	Information Retrieval
Jiawei Han	Data Mining
Steve LaValle	Algorithms and Robotics
Jeff Erickson	Computational Geometry
David Forsyth	Computer Vision

Your choice

Name	Area
Dan Roth	Machine Learning and NLP
Chengxiang Zhai	Information Retrieval
Jiawei Han	Data Mining
Steve LaValle	Algorithms and Robotics
Jeff Erickson	Computational Geometry
David Forsyth	Computer Vision

How about a table of size 26?

0 (A)	
1 (B)	
...	
5 (E)	→ Computational Geometry
6 (F)	→ Computer Vision
7 (G)	
8 (H)	→ Data Mining
9 (I)	
...	

Let me guess, you used a Dictionary ADT and can support insert, find, and remove in $O(\log n)$ time?

That's pretty good, but what if you could do it in constant time? Look at the data again, and try to find some sneaky, "cheating" way to look up an area given a name.

A perfect hash function

- You just wrote a perfect hash function!
- $hash(name) = \text{first char of last name} - 'A'$
- A perfect hash function is a bijection between the keyspace and a collection of small integers
 - Each key maps to a different int (one-to-one)
 - Given 26 different last names, each name hashes on the entire sequence of ints (onto)
- Well, that was contrived, but can we use this general idea to create a fast implementation of the dictionary ADT?
 - Yes, of course!

Collisions

- Given a dataset $\{200, 205, 210, 215, 220, \dots, 595, 600\}$, and a table of size $N = 100$, create a hash function
 - How about $h(k) = k\%N$?
- A collision is when two different keys hash to the same location in the table
 - That is, $k_1 \neq k_2, h(k_1) = h(k_2)$
- Is our table too small? Well, there are 80 keys and 100 cells...
 - Better hash function: $h(k) = \frac{k-200}{5}$
- How will this work for other datasets?

Hash functions...

- Consist of two parts
 - 1 A hash: function mapping a key k to an integer i
 - 2 A compression: function mapping i onto array cells 0 to $N - 1$
- SUHA: Simple Uniform Hashing Assumption
 - $\forall k_1, k_2 : P(h(k_1) = h(k_2)) = \frac{1}{N}$
- Choosing a hash function is tricky
 - Don't create your own!
 - Smart people make dumb hash functions (Knuth's multiplicative hash)
- What is a bad hash function?
 - Lots of collisions
 - Non-constant computation time
 - Nondeterministic

Good hash functions

Based on the previous list, good hash functions should satisfy the following:

- 1 Computed in $O(1)$ time
- 2 Deterministic: $k_1 = k_2 \rightarrow h(k_1) = h(k_2)$
- 3 Satisfy the simple uniform hashing assumption (SUHA)

Making a hash function

Say we want to map phone numbers (xxx-xxx-xxxx) to names.

Consider the following choices for hashing phone numbers:

- First 3 digits mod N
- Last 3 digits mod N
- Sort the numbers, then take the first 3 digits mod N
- Randomly select 3 digits, combine, and mod N

Hashing an object

We have the following Point class:

```
struct Point
{
    int x;
    int y;
    int z;
};
```

Consider the following hash functions for hashing a Point P :

- $h(P) = P.x \% N$
- $h(P) = (P.x + P.y + P.z) \% N$
- $h(P) = \max(P.x, P.y, P.z) \% N$
- $h(P) = (31^0 \cdot P.x + 31^1 \cdot P.y + 31^2 \cdot P.z) \% N$

Hashing in C++

As we've mentioned before, the C++ Standard Template Library (STL) has a wide array of data structures and algorithms. It has a tree-based Dictionary (`std::map`) that is implemented with a Red-Black tree. It also has a hash table (`std::unordered_map`). (Why is it called this?)

- `std::unordered_map` already has hash functions for some basic types like `int`, `std::string`, and `double`
- If you want to include your own objects in a `std::unordered_map`, you have to write your own implementation of a templated `hash<T>` function that returns a `size_t`

Hashing in Java

Like C++, Java has its own tree-based dictionary (`java.util.TreeMap`) using Red-Black trees. It also has its own hash table, `java.util.HashMap`, which you have probably heard of.

- All Java objects inherit a `hashCode()` function, which returns an `int`. Like C++, this is already implemented for some basic types
- You need to implement this function if your object is to be used in a `HashMap` or similar structure!

Hash Tables

A hash table is a Dictionary ADT implementation that makes use of hash functions to map (key, value) pairs to slots in an array. A hash table consists of:

- 1 An array ✓
- 2 A hash function ✓
- 3 A collision resolution strategy (?)

Using Hash Tables

Hash tables are useful because they offer amortized $O(1)$ running times for the dictionary operations. This running time comes from SUHA, as we will investigate next lecture.

Let's think of some applications of hash tables (or dictionaries in general). If using a hash table, what hash function would you use?

- How can you efficiently store a large, sparse matrix?
- How can you keep track of whether a certain IP has visited a site before?
- How can you make a histogram of words from a text file?

lecture23: Hash Tables

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

18th July, 2013

Announcements

- mp6.1 extra credit due tomorrow tonight (7/19)
- lab_avl due tonight
- mp6 due Monday (7/22)



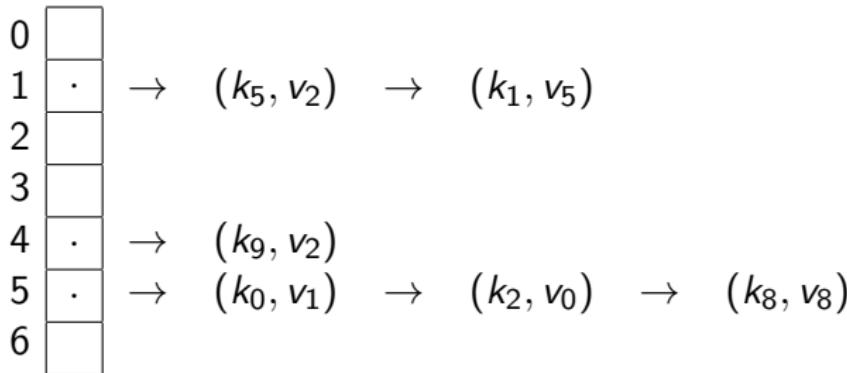
Hash tables

A hash table is a Dictionary ADT implementation that makes use of hash functions to map (key, value) pairs to slots in an array. A hash table consists of:

- 1 An array ✓
- 2 A hash function ✓
- 3 A collision resolution strategy (?)

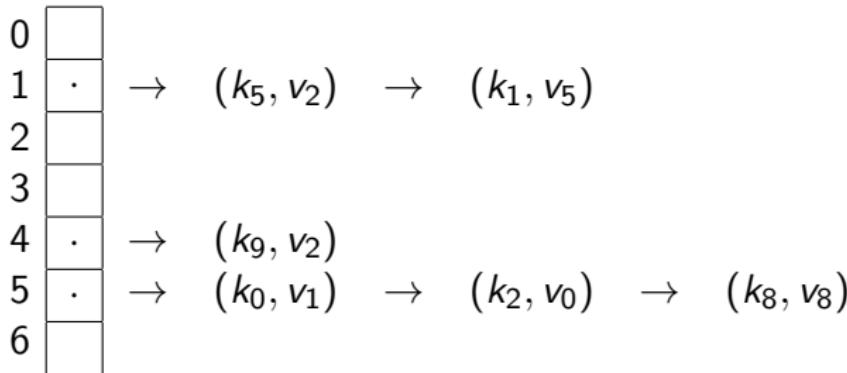
Separate chaining

Strategy: keep a linked list (or other structure) at each index in the array. If two or more keys hash to the same index, add them into the linked list. How do find, insert, and remove work?



Separate chaining

Define the **load factor** to be the number of elements stored in the hash table divided by the number of buckets: $\alpha = \frac{n}{N}$. We want to keep α approximately constant. How can we do this?



Resizing

- When α becomes too large (usually chosen to be between .66 and .75), we need to increase N
- This means doubling the table size and *rehashing each element*
- Why is it necessary to rehash each element? Can't we just copy them over to corresponding cells in the larger table?

SUHA and hash table running times

We've mentioned previously that all dictionary ADT operations are constant (or amortized constant) time. Let's see how SUHA and table resizing can ensure this is true.

Operation	Worst Case	SUHA
insert	$O(1)^*$	$O(1)^*$
successful remove/find	$O(n)$	$O(\frac{1}{2}(\frac{n}{N})) = O(\frac{\alpha}{2}) = O(1)$
unsuccessful remove/find	$O(n)$	$O(\frac{n}{N}) = O(\alpha) = O(1)$

Recall we are using linked lists as our "buckets". Could we use some other data structure? How (if at all) would that change our running times?

Separate chaining: insert

Linear probing in some senses is simpler than separate chaining.

insert: if a key is already in the desired spot, keep moving along the table until you find an empty spot and put it there!

0	
1	
2	
3	
4	
5	
6	

From the previous example, we know k_5 and k_1 have the same hash as well as k_0, k_2 , and k_8 .

Let's insert in this order: k_0, k_1, k_2, k_5

Make note $h(k_0) = 5, h(k_1) = 1, h(k_2) = 5, h(k_5) = 1$

Linear probing: insert

Linear probing in some senses is simpler than separate chaining.
insert: if a key is already in the desired spot, keep moving along the table until you find an empty spot and put it there!

0	
1	(k_1, v_5)
2	(k_5, v_2)
3	
4	
5	(k_0, v_1)
6	(k_2, v_0)

From the previous example, we know k_5 and k_1 have the same hash as well as k_0 , k_2 , and k_8 .

Let's insert in this order: k_0, k_1, k_2, k_5

Make note $h(k_0) = 5, h(k_1) = 1, h(k_2) = 5, h(k_5) = 1$

Linear probing: find

For find, we need to hash the key as usual, and look up that index in the hash table.

0	
1	(k_1, v_5)
2	(k_5, v_2)
3	
4	
5	(k_0, v_1)
6	(k_2, v_0)

Let's call `find` on these objects: k_0, k_5, k_6 . Note $h(k_6) = 1$.

If the key we're looking for is not in the cell we hash to, we keep looking (probing) until we either find it or get to a completely empty cell.

Linear probing: remove

When removing elements, we need to mark the cell it was in to indicate something was there at one point. Why is this necessary in order for `find` to work properly?

0	
1	X
2	(k_5, v_2)
3	
4	
5	(k_0, v_1)
6	(k_2, v_0)

Let's remove k_1 , marking its cell that something existed there.

Now, when we call `find` on k_5 , we know to keep looking past cell 1, since there used to be something there.

Clustering

- In linear probing, clusters of keys are likely to appear as keys hash into a similar range and have to traverse to adjacent indices
- Hashing into a cluster becomes more and more likely as more data is inserted
- This increases the cost of insert!



Solutions to clustering

- Quadratic hashing
 - If the hashed cell index is occupied, advance forward a number of steps determined by a quadratic polynomial
 - For example, if the initial hash is i , advance in steps $(i, i + 1^2, i + 2^2, i + 3^2, i + 4^2, \dots)$
- Double hashing
 - Uses two hash functions, h_1, h_2
 - If the cell $h_1(k)$ is occupied, advance in a fixed step size determined by $h_2(k)$ until there is an open cell
- Cuckoo hashing
 - Uses two hash functions, h_1, h_2
 - Use h_1 to insert. If there is a collision, “kick out” the offender and rehash it using h_2
 - Repeat this process until an open cell is found or an infinite loop is encountered

LP performance

insert, find, and remove expected number of probes for linear probing:

successful:

$$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

unsuccessful:

$$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)^2$$

Don't memorize these! Just know that they are all constant assuming α is held constant.

Open vs closed hashing

- Separate chaining is an example of *open hashing*: the actual objects live “outside” the hash table
 - Variants use different data structures as the buckets
- Linear probing is an example of *closed hashing*: the objects live inside the hash table (array) itself
 - Variants use *non-linear* probing strategies; this is in hopes to disperse the keys more throughout the table and avoid clustering

Philosophical questions

Why do we even have balanced BSTs if hash tables are so awesome?

- insert, find, delete running times
- sort running times
- Memory usage
- operator`<` vs a hash function
- Complexity (not in the running time sense)

Denial of Service via Algorithmic Complexity Attacks

- Paper by Crosby and Wallach in 2003, currently cited by 215
- They explicitly made the worst case happen every time in order to slow down (and break) operations
 - Intrusion detection system: send specific packets to server, using very low bandwidth
 - Perl: insert specific strings into an associative array
 - Linux kernel (2.4.20): save files with specific names

lecture24: Disjoint Sets

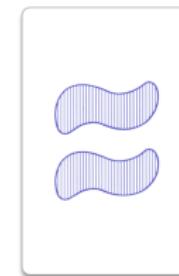
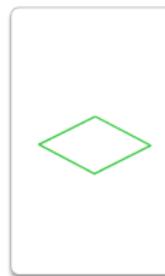
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

22nd July, 2013

Announcements

- mp6 due tonight
- mp7 out soon!
- mt3 tomorrow night (7/23)
- Optional review instead of lab tomorrow



Code Challenge Friday Night! 6pm in the lab

Braintree



A New Task

Given the set $\{C\#, C++, Java, Ruby, Python\}$, partition students by their favorite programming language. Each student has an integer ID (we can keep track of this mapping with a Dictionary).

What kind of operations can we perform?

- `find(4)`
 - What does this return?
- `find(3) == find(1)?`
- if `find(2) != find(0)`, then `union(2, 0)`

We're modeling equivalence relations

What is an equivalence relation again?

A given binary relation \sim on a set S is said to be an equivalence relation if and only if it is reflexive, symmetric and transitive.

Equivalently, $\forall a, b, c \in S$:

- $a \sim a$ (reflexive)
- $a \sim b \rightarrow b \sim a$ (symmetric)
- $a \sim b \wedge b \sim c \rightarrow a \sim c$ (transitive)

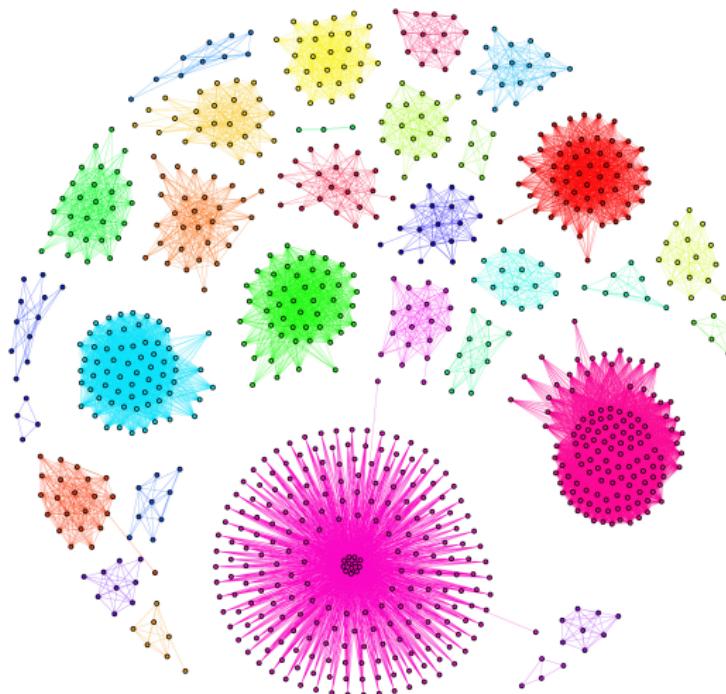
How does this translate to our programming language example?

Another example

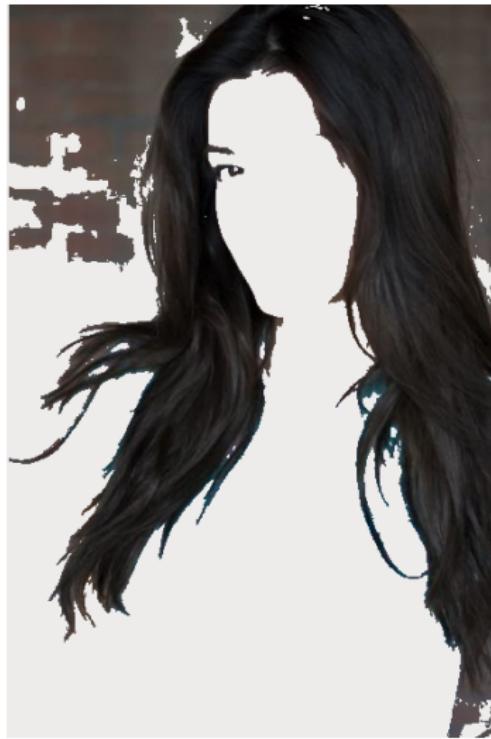
How could you model a social network using this framework?
What is the equivalence relation?

What about RGBAPixels in a PNG?

Partitioning a Social Network



Partitioning Regions in an Image



Our New ADT

We need a **union-find ADT**: Disjoint Sets. It supports

- 1 `addelements`, to initialize the structure: *initially all elements are in their own set*
- 2 `find`, to return which set an element belongs to (*by returning a representative element from the set*)
- 3 `union`, to connect two elements together

```
class DisjointSets {  
    public:  
        void addelements(int num);  
        int find(int id);  
        void union(int id1, int id2);  
    private:  
        // ???  
};
```

Our first implementation

```
DisjointSets dsets;  
dsets.addelements(8);
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

```
dsets.union(2, 5);
```

0	1	2	3	4	5	6	7
0	1	2	3	4	2	6	7

```
dsets.union(5, 7);  
dsets.union(0, 1);  
dsets.union(0, 6);
```

0	1	2	3	4	5	6	7
1	6	2	3	4	2	1	2

- When we create the set, each element is its own set and its own representative
- When unioning, choose an arbitrary element as the set representative
- How does find work?
- What if we wanted to do $\text{union}(1, 2)$?
- What is the running time of find and union?

DisjointSets via an array

It seems our array implementation gives great running times for `find`, but `union` can be very inefficient: $O(n)$ when combining large sets together.

We should think of a way to get sublinear running times for our `DisjointSets` operations...

UpTrees: A better DS implementation

- We can imagine our sets as a collection of trees (Uptrees) instead of elements in an array
- The root of each tree is the representative of the set it is in
- When unioning two elements, we can just point the root at the set we're unioning with!
- We don't even have to use pointers for Uptrees – instead, each element's field holds the index of its parent (roots can have -1 as their data)
- Let's draw what this looks like...

Imagining Uptrees

```
DisjointSets dsets;  
dsets.addelements(8);
```

0	1	2	3	4	5	6	7
-1	-1	-1	-1	-1	-1	-1	-1

```
dsets.union(2, 5);
```

0	1	2	3	4	5	6	7
-1	-1	5	-1	-1	-1	-1	-1

```
dsets.union(5, 7);  
dsets.union(0, 1);  
dsets.union(0, 6);
```

0	1	2	3	4	5	6	7
1	6	5	-1	-1	7	-1	-1

- When unioning, we need to deal with the roots to make sure all elements are unioned correctly
- By default, let's point the root of the first argument to the root of the second
- Can you draw the uptrees?

Concept Check: Draw the Uptrees

Draw the forest of Uptrees if this is our array.

0	1	2	3	4	5	6	7
-1	3	-1	-1	6	3	0	2

Could you write code that generated this forest? How many disjoint sets are there?

Our New DS Implementation

```
// return the root of the Uptree containing "id"
int DisjointSets::find(int id)
{
    // ... implementation ...
    return root;
}

// combine the sets containing id1 and id2
void DisjointSets::union(int id1, int id2)
{
    // ... implementation ...
}
```

Analyzing DS

- The running time of DS operations using Uptrees depends on...
- So the worst case running time is still $O(n)$
- What does an ideal Uptree look like?
- If we have to union two Uptrees together, how can we do it so we don't increase the height?

Smart Unions

- Union by height
 - Keeps the overall height of the tree as short as possible
- Union by size
 - Increases the distance to the root for the *fewest* number of nodes
- Both of these smart union schemes guarantee a $O(\log n)$ uptree height in all the disjoint sets

Union By Size

```
// combine the sets containing id1 and id2
void DisjointSets::union(int id1, int id2)
{
    int root1 = find(id1);
    int root2 = find(id2);
    int newSize = _set[root1] + _set[root2]; // ??

    // root1 has more elements
    if(_set[root1] < _set[root2])
    {

    }
    // root2 has more elements (or same)
    else
    {

    }
}
```

Union By Height

- We think you should be able to write a union-by-height implementation, too
- (In mp7, we'll assume you write a union-by-size though)
- What other considerations do we have to take if doing union-by-height?

Another Optimization

- After computing the root in `find`, set the current node's parent to be that root!
- This makes our trees much *shorter*, which is great because we already know all the operations on our disjoint sets are $O(h)$

```
int DisjointSets::find(int id)
{
    }
```

Union-find with smart unions and path compression

New running times...

$\log^* n \equiv$ number of times you can take the logarithm of n before you get to 1

n	$\log^* n$
1	0
2	1
4	2
16	3
65,536	4
$2^{65,536}$	5

In any sequence of m union and find operations on a collection of n items, the running time is $O(m \log^* n)$ (effectively constant).

Using DS

- mp7 (how?)
- lab_graphs: Kruskal's Algorithm
- lots of other applications...

lecture25: Graphs

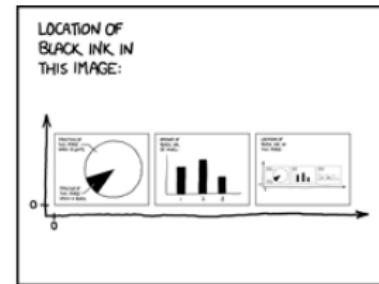
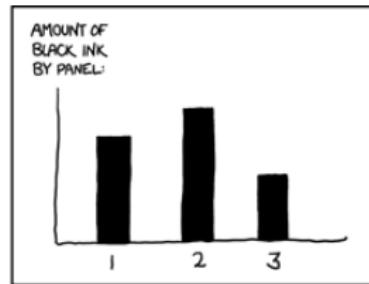
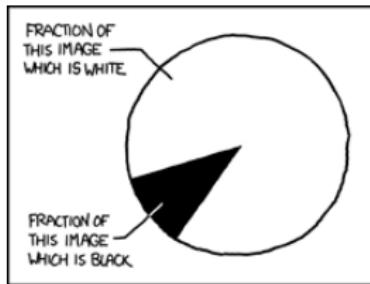
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

23rd July, 2013

Announcements

- mt3 tonight at 7pm
- mp7.1 extra credit due Friday night (7/26) at 11pm



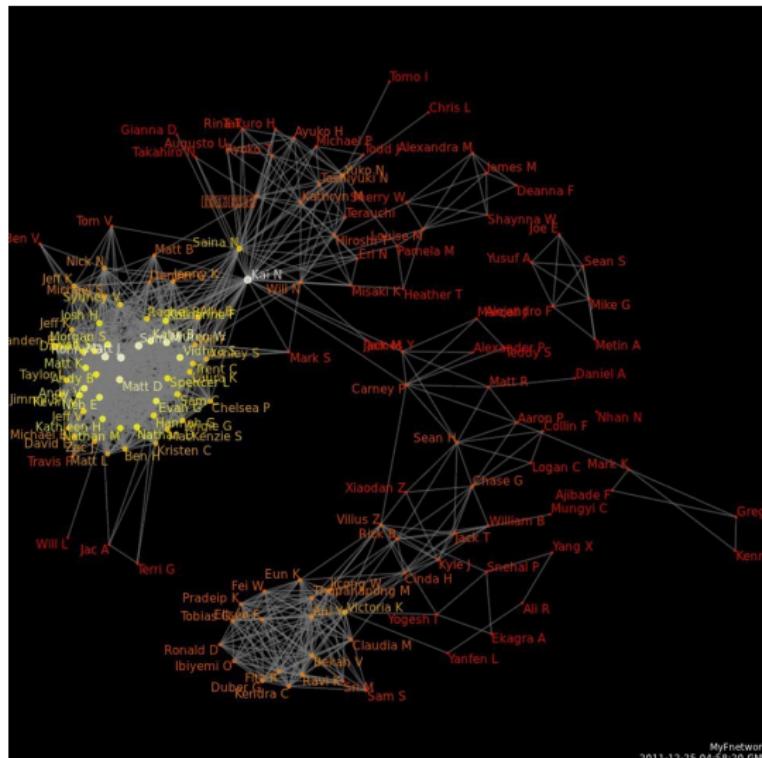
Exciting Mathematical Definition

A graph $G = (E, V)$ is defined to be a set of edges E and a set of vertices V :

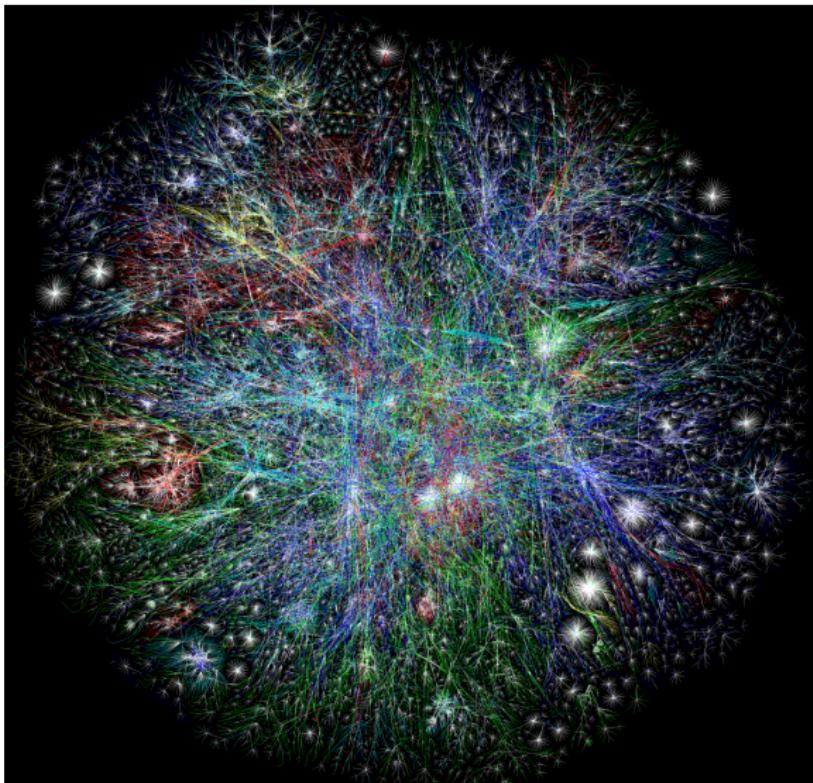
- $V = \{v_0, v_1, v_2, \dots, v_n\}, |V| = n$
- $E = \{e_0, e_1, e_2, \dots, e_m\}, |E| = m$
- $e_i = (v_j, v_k), i \in [0, m], j, k \in [0, n]$, that is, each edge is a pair of two vertices

We will use graphs to model many problems by giving some specific meaning to vertices (also called nodes) and edges.
A graph can be thought of as a generalization of a tree.

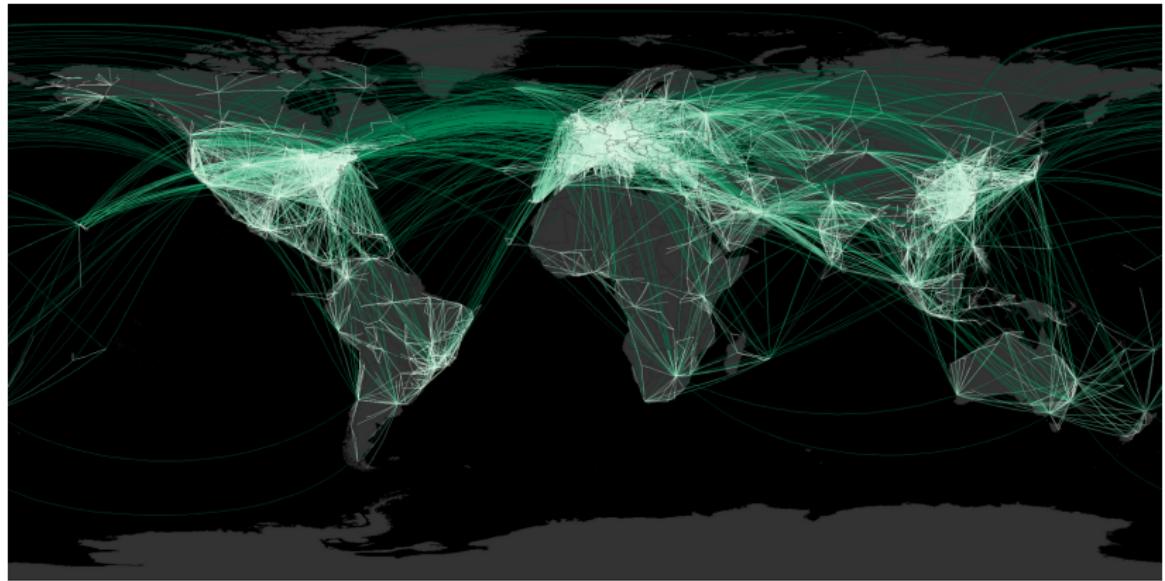
A social network



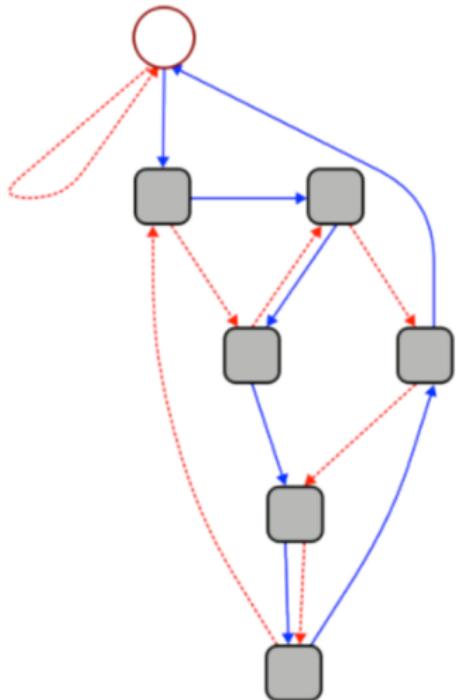
A weighted graph



A weighted, directed graph

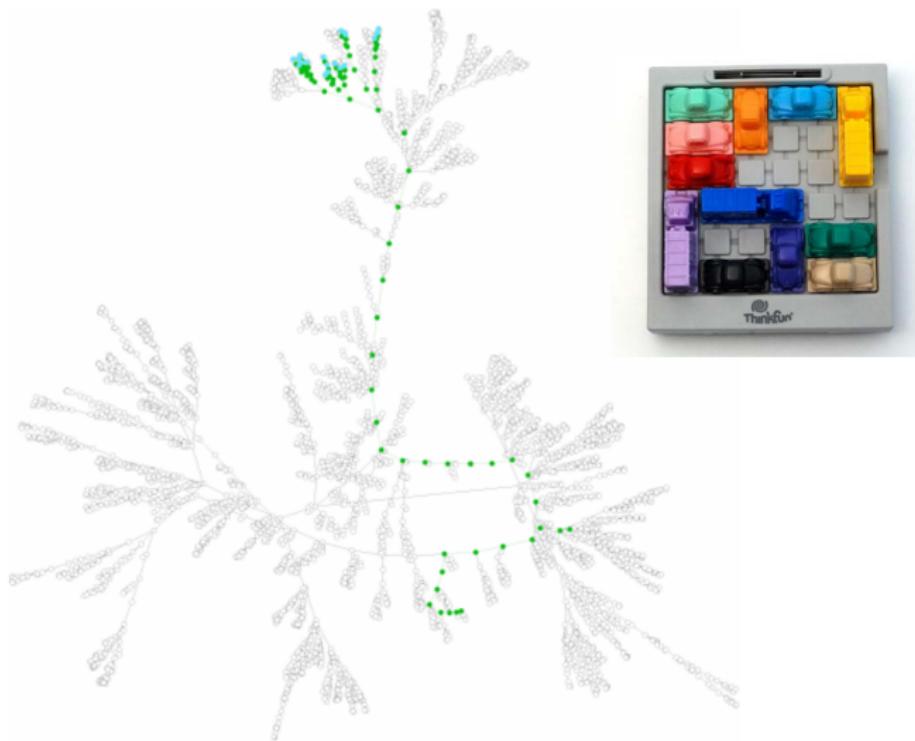


A state machine



- 1 Start at the circle node at the top
- 2 For each digit d in the given number, follow d blue (solid) edges in succession. As you move from one digit to the next, follow 1 red (dashed) edge.
- 3 If you end up back at the circle node, your number is...

All possible moves...



More Graphs!

Graph Vocab

We already talked about...

- weighted *vs* unweighted
- directed *vs* undirected

There's also...

- connected *vs* unconnected
- simple *vs* ...not simple?
 - A simple graph has no self-loops and no multi-edges

Graph Functions

- $\text{incidentEdges}(v) = \{(v, v') | v' \in E\}$
 - all edges containing a given vertex
- $\text{degree}(v) = |\text{incidentEdges}(v)|$
 - the number of incident edges for a given vertex
- $\text{adjacent}(v) = \{v' | (v, v') \in E\}$
 - the vertices directly connected to a given vertex

More Graph Vocab

Identify each of these on our graphs

- Path
- Cycle
- Subgraph
- Connected component (directed only)
- Acyclic graph
- Spanning tree

How many edges?

- There are at least...
 - If connected, $|E| \geq n - 1$
 - If not connected, $|E| \geq 0$
- There are at most...
 - If simple, $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}$
 - If not simple, $|E| \in [0, \infty)$

A useful formula:

$$\sum_{v \in V} \text{degree}(v) = 2m$$

Why is this true?

Graph Problems

- Finding a minimum spanning tree (weighted graphs only):
 - cluster analysis, handwriting recognition, circuit design, broadcasting in networks, analyzing gene expressions
- Finding strongly connected components
 - recommendation systems, formal verification model checking, graph compression
- Shortest path... obvious applications!
- Hamiltonian path
 - travelling salesman, knight's tour, reductions
- Graph coloring
 - sudoku solver, scheduling, CPU register allocation, bandwidth allocation
- Mincut-Maxflow
 - course scheduling, dependency resolution, determining whether a team is eliminated from qualifying in playoffs, image segmentation, staff time allocation

Minimum Spanning Trees

How should we connect all these routers with the minimum total latency?

Shortest Path

What is the fewest number of layovers on a trip?

Shortest Path

What is the shortest distance from one point to another?

Graph Coloring

How can we pair up teams for a tournament so there are no conflicts?

lecture26: Graph Implementations

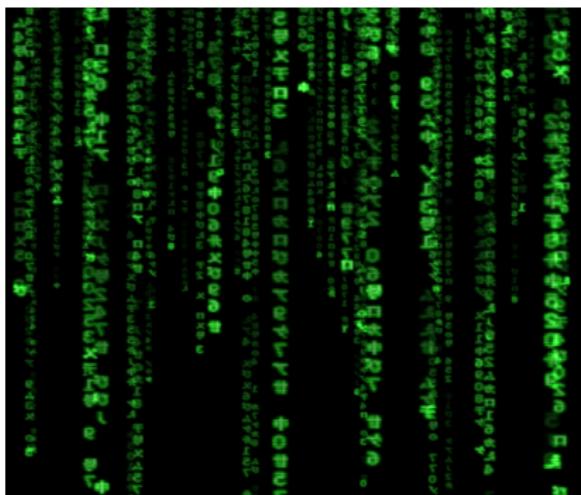
Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

24th July, 2013

Announcements

- mp7.1 extra credit due Friday (7/26) at 11pm
- Code challenge Friday (7/26) night!
- lab_hash tomorrow, due Saturday night (7/27)



Our Graph ADT... our *last* ADT :'(

```
// assume we have a Vertex and Edge class, each templated
// to hold an arbitrary object

class Graph {
public:
    void insertVertex(const Vertex & v);
    void insertEdge(const Edge & e, Vertex v1, Vertex v2);
    void removeVertex(const Vertex & v);
    void removeEdge(const Edge & edge);

    vector<Edge> getIncident(const Vertex & v) const;
    vector<Vertex> getAdjacent(const Vertex & v) const;
    bool adjacent(const Vertex & v1, const Vertex & v2) const;

private:
    // ???
};
```

Using a graph

```
Graph socialNet;

Vertex<Person> v1(Person("John"));
Vertex<Person> v2(Person("Elliot"));
Vertex<Person> v3(Person("Ted"));
Vertex<Person> v4(Person("Perry"));

socialNet.insertVertex(v1);
socialNet.insertVertex(v2);
socialNet.insertVertex(v3);
socialNet.insertVertex(v4);

socialNet.insertEdge(.92, v1, v2);
socialNet.insertEdge(.22, v2, v4);
socialNet.insertEdge(.12, v1, v4);
```

Our first try: an “Edge List”

Graph building with an Edge list

How do these work and what are the running times?

```
void insertVertex(const Vertex & v);  
void insertEdge(const Edge & e, Vertex v1, Vertex v2);  
void removeVertex(const Vertex & v);  
void removeEdge(const Edge & edge);
```

Graph functions with an Edge list

How do these work and what are the running times?

```
vector<Edge> getIncident(const Vertex & v) const;  
vector<Vertex> getAdjacent(const Vertex & v) const;  
bool adjacent(const Vertex & v1, const Vertex & v2) const;
```

Graph building with an adjacency matrix

Graph building with an adjacency matrix

How do these work and what are the running times?

```
void insertVertex(const Vertex & v);  
void insertEdge(const Edge & e, Vertex v1, Vertex v2);  
void removeVertex(const Vertex & v);  
void removeEdge(const Edge & edge);
```

Graph functions with an adjacency matrix

How do these work and what are the running times?

```
vector<Edge> getIncident(const Vertex & v) const;  
vector<Vertex> getAdjacent(const Vertex & v) const;  
bool adjacent(const Vertex & v1, const Vertex & v2) const;
```

Graph building with an adjacency list

Graph building with an adjacency list

How do these work and what are the running times?

```
void insertVertex(const Vertex & v);  
void insertEdge(const Edge & e, Vertex v1, Vertex v2);  
void removeVertex(const Vertex & v);  
void removeEdge(const Edge & edge);
```

Graph functions with an adjacency list

How do these work and what are the running times?

```
vector<Edge> getIncident(const Vertex & v) const;  
vector<Vertex> getAdjacent(const Vertex & v) const;  
bool adjacent(const Vertex & v1, const Vertex & v2) const;
```

Graphs Performance

Given a simple graph with $|V| = n, |E| = m...$

	Edge List	Adj Matrix	Adj List
Space <code>insertVertex(v)</code> <code>removeVertex(v)</code> <code>insertEdge(v,w)</code> <code>removeEdge(v,w)</code> <code>getIncident(v)</code> <code>getAdjacent(v)</code> <code>adjacent(v,w)</code>			

lecture27: Graph Traversals

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

25th July, 2013

Announcements

- mp7.1 extra credit due tomorrow night (7/26)
- Code challenge tomorrow night (7/26) at 6pm in 0224
- lab_hash due Saturday night (7/27)
- This is our last Thursday class

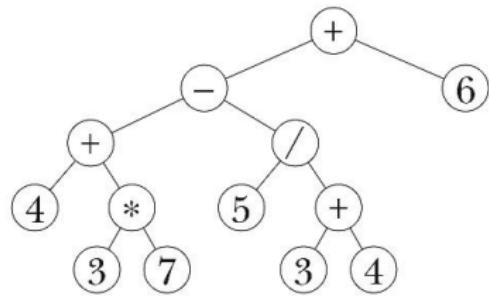


Traversals

- We know how to do traversals on trees, but now we want to traverse a graph
- That is, we want to visit every vertex and every edge in the graph, *honoring the connectivity of the graph*
- Compared to tree traversals...
 - there is no obvious start point
 - there is no obvious order
 - there is no obvious way to denote progress... how can we make sure we aren't traversing in circles forever??

A level order traversal

```
template <class T>
void BinaryTreeNode::traverse() const
{
    Queue<TreeNode*> q;
    q.enqueue(root);
    while(!q.empty())
    {
        TreeNode* cur = q.dequeue();
        if(cur != NULL)
        {
            yell(cur->data);
            q.enqueue(cur->left);
            q.enqueue(cur->right);
        }
    }
}
```

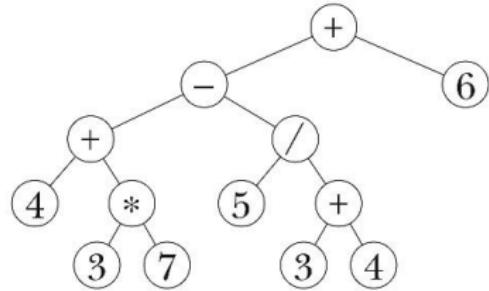


Prints:

+ - 6 + / 4 * 5 + 3 7 3 4

A level order traversal Breadth-First Search

```
template <class T>
void BinaryTree<T>::traverse() const
{
    Queue<TreeNode*> q;
    q.enqueue(root);
    while(!q.empty())
    {
        TreeNode* cur = q.dequeue();
        if(cur != NULL)
        {
            yell(cur->data);
            q.enqueue(cur->left);
            q.enqueue(cur->right);
        }
    }
}
```



First, explore all nodes at a distance one away, then at a distance two, then, three.....

BFS

- From the start point, expands outwards like a ripple in a pond
- Every time we see an unvisited node, this is the *shortest distance* from the start to that node (by number of edges)
- BFS is written most intuitively in an iterative fashion using a Queue as an ordering structure
- BFS can use the labels **discovery** and **cross** to label edges

BFS pseudocode

- 1 Enqueue the start node
- 2 Dequeue a node and enqueue any adjacent nodes that have not yet been visited
- 3 If the queue is empty, every node in the graph has been examined; stop
- 4 Otherwise, if the queue is not empty, repeat from step 2

BFS example

BFS implementation

Assume we are given the graph and some start vertex.

```
void BFS(Graph & graph, const Vertex & start)
{
    queue<Vertex> q;
    q.push(start);
    graph.setVertexLabel(start, "VISITED");

    while(!q.empty())
    {
        // traverse!
    }
}
```

How could we modify this to search for a specific vertex, or to work on a graph that isn't connected?

BFS implementation: the traversal

```
while(!q.empty())
{
    Vertex v = q.front();
    q.pop();

    vector<Vertex> adj = graph.getAdjacent(v);
    for(size_t i = 0; i < adj.size(); ++i)
    {
        if(graph.getVertexLabel(adj[i]) == "UNEXPLORED")
        {

        }
        else if(graph.getEdgeLabel(v, adj[i]) == "UNEXPLORED")
        {

        }
    }
}
```

BFS analysis

- What is the running time of BFS in terms of n and m ?
- ...does it change based on our graph implementation?
- How can we use BFS to make a spanning tree?
- How can we use BFS to detect cycles?

DFS

- From the start point, DFS explores as far as possible along each branch, then starts *backtracking*
- DFS is written intuitively in *either* an iterative fashion (very similar to BFS) or recursively
- DFS can use the labels **discovery** and **back** to label edges

DFS pseudocode (using a stack)

- 1 Push the start node
- 2 Pop a node and push any adjacent nodes that have not yet been visited
- 3 If the stack is empty, every node in the graph has been examined; stop
- 4 Otherwise, if the stack is not empty, repeat from step 2

DFS example

DFS implementation (iterative)

Assume we are given the graph and some start vertex.

```
void DFS(Graph & graph, const Vertex & start)
{
    stack<Vertex> s;
    s.push(start);
    graph.setVertexLabel(start, "VISITED");

    while(!s.empty())
    {
        // traverse!
    }
}
```

How could we modify this to search for a specific vertex, or to work on a graph that isn't connected?

DFS implementation (iterative): the traversal

```
while(!s.empty())
{
    Vertex v = s.top();
    s.pop();

    vector<Vertex> adj = graph.getAdjacent(v);
    for(size_t i = 0; i < adj.size(); ++i)
    {
        if(graph.getVertexLabel(adj[i]) == "UNEXPLORED")
        {

        }
        else if(graph.getEdgeLabel(v, adj[i]) == "UNEXPLORED")
        {

        }
    }
}
```

DFS implementation (recursive)

```
void DFS(Graph & graph, Vertex cur)
{
    graph.setVertexLabel(cur, "VISITED");

    vector<Vertex> adj = graph.getAdjacent(cur);
    for(size_t i = 0; i < adj.size(); ++i)
    {
        if(graph.getVertexLabel(adj[i]) == "UNEXPLORED")
        {

        }
        else if(graph.getEdgeLabel(cur, adj[i]) == "UNEXPLORED")
        {

        }
    }
}
```

DFS analysis

- What is the running time of DFS in terms of n and m ?
- ...does it change based on our graph implementation?
- How can we use DFS to make a spanning tree?
- How can we use DFS to detect cycles?

Bipartite graphs

Bipartite graph test

lecture28: Minimum Spanning Trees

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

29th July, 2013

Outline

1 Announcements

2 MSTs

3 Kruskal's

4 Prim's

Announcements

- mp7 due tonight!
- lab_graphs out tomorrow,
due Thursday, 8/1
- final exam this Friday (8/2),
10:30am-12:30pm in this
room



Outline

1 Announcements

2 MSTs

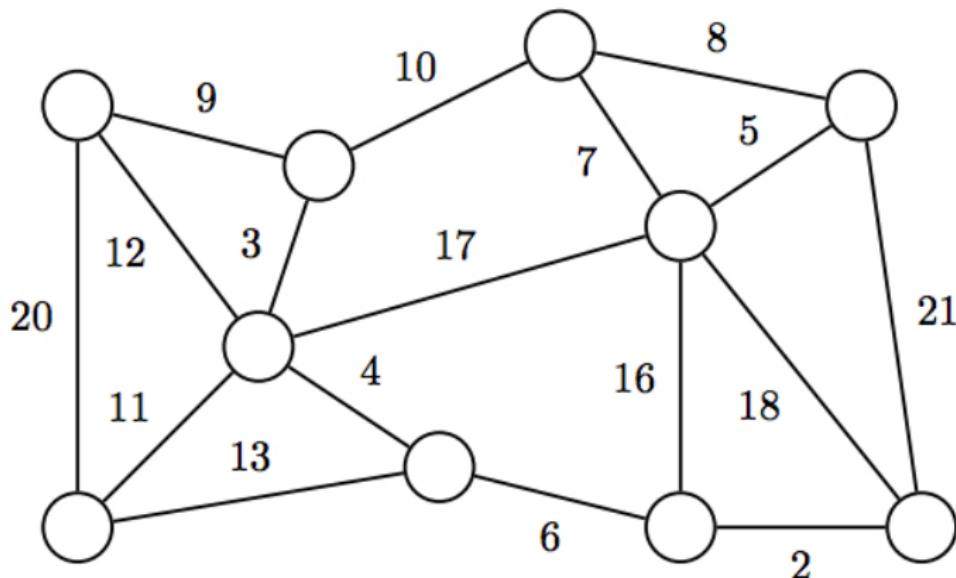
3 Kruskal's

4 Prim's

MST definition

- A **spanning tree** T of a graph G connects all vertices together with no cycles (hence a tree)
- If we define the weight of a spanning tree to be the sum of its edge weights, a *minimal* spanning tree has a minimal weight for a given graph
- We'll investigate two algorithms to find the MST of a graph: Kruskal's algorithm and Prim's algorithm
- We'll assume both these algorithms run on weighted, undirected graphs (it's easy enough to modify them to work on directed graphs)

Example graph



Outline

1 Announcements

2 MSTs

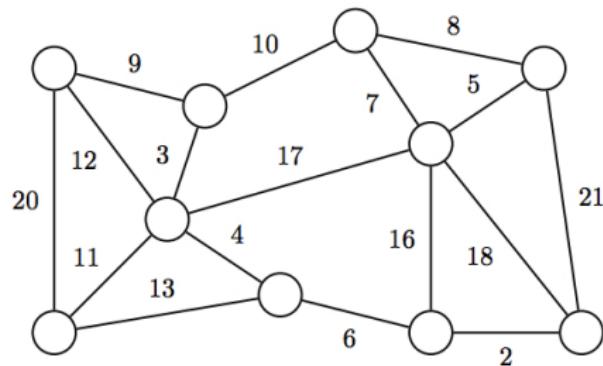
3 Kruskal's

4 Prim's

Kruskal's Pseudocode

- 1 Initialize our output graph, $T = (V', E')$, $V' = V$, $E' = \{\}$
- 2 Initialize a disjoint sets structure S , where each vertex represents a set (all in own set to begin with)
- 3 Initialize a priority queue, P , holding all the **edges** in the original graph
- 4 $e = P.\text{removeMin}()$
 - If e connects two vertices from different sets, add e to E' , and union the two vertices from e in S
 - If e connects two vertices from the same set, do nothing (this would create a cycle)
- 5 Repeat step 4 until $|E'| = n - 1$

Running Kruskal's



Kruskal's Analysis

Here's the outline of Kruskal's:

- Initialize disjoint sets
- Initialize priority queue
- Call `removeMin` $n - 1$ times:
 - Call `union` if necessary

	Binary Heap	Sorted Array
<code>build</code> each <code>removeMin</code>		

Total time using a binary heap:

Total time using a sorted array:

Outline

1 Announcements

2 MSTs

3 Kruskal's

4 Prim's

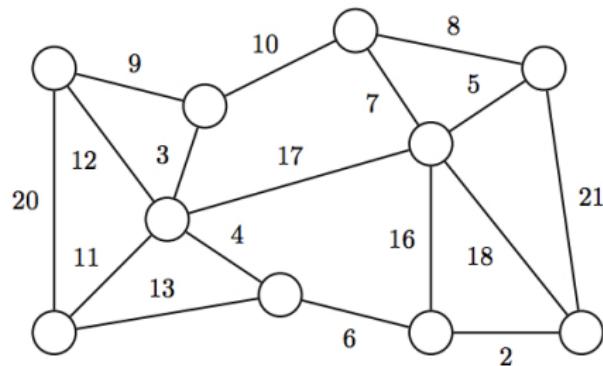
Prim's Pseudocode

- 1 Initialize a priority queue P , to hold **vertices** based on a “cost” (initially all ∞); change a start vertex’s cost to 0
- 2 Initialize an empty dictionary D : vertex \rightarrow parent
- 3 $v = P.\text{removeMin}()$, label v as visited
 $\forall w \in \text{adjacent}(v)$:
 - if w is unvisited and $\text{cost}(v, w) < P[w]$
 - $P.\text{decreaseKey}(w, \text{cost}(v, w))$
 - $D[w] = v$
- 4 Repeat step 3 for n times (until P is empty), then create T by using the parents in D

DecreaseKey

- `decreaseKey` is a possible priority queue ADT function
- It assumes we have direct access to objects inside the PQ
- How would you write `decreaseKey` for a binary heap, given an index?
- This means the running time for `decreaseKey` in a binary heap is...

Running Prim's



Prim's Analysis

Here's an outline of Prim's (assuming D is a hash table):

- Initialize priority queue
- Call `removeMin` n times:
 - Call `getAdjacent`
 - Call `decreaseKey` if necessary

	Binary Heap	Fibonacci Heap
build		$O(n)$
each <code>removeMin</code>		$O(\log n)*$
each <code>decreaseKey</code>		$O(1)*$

But how many times do we call `decreaseKey`?

$$\sum_{v \in V} \deg(v) = 2m = O(m)$$

Prim's Analysis II

We call `build` once, `removeMin` n times, `getAdjacent` n times, and `decreaseKey` $2m$ times.

Using all this information, we can build the final running time for Prim's based on the following:

$$O(\text{build} + n \cdot (\text{removeMin} + \text{getAdjacent}) + m \cdot \text{decreaseKey})$$

You should be able to plug in running times (and simplify) using any PQ structure and any graph implementation! Also remember for connected graphs:

$$n - 1 \leq m \leq n^2$$

Example running times

Find the running time for Prim's using...

- adjacency list and binary heap
- adjacency matrix and binary heap
- adjacency matrix and Fibonacci heap
- adjacency list and sorted array

What's the best running time for Prim's you can build?

lecture29: Shortest Path Algorithms

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

30th July, 2013

Outline

1 Announcements

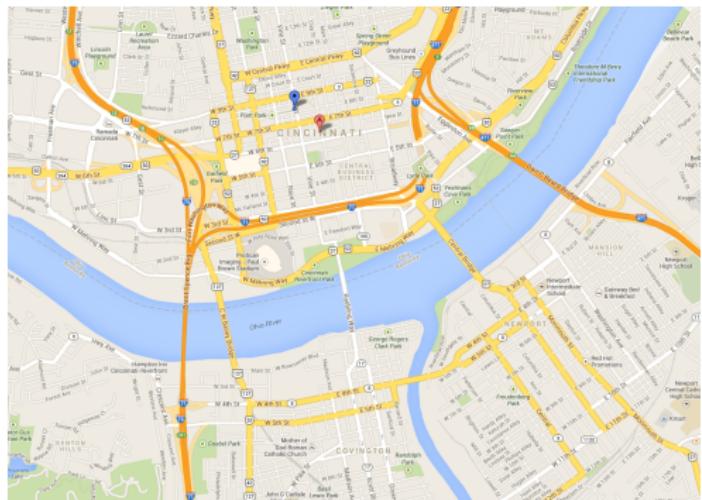
2 MST Review

3 Dijkstra's

4 A^*

Announcements

- lab_graphs due Thursday, 8/1
- final exam this Friday (8/2), 10:30am-12:30pm in this room



Why?

```
if(boolean_statement_or_condition())
    return true;
else
    return false;
```

Why?

```
bool ret;  
  
if(boolean_statement_or_condition())  
    ret = true;  
else  
    ret = false;  
  
return ret;
```

Outline

1 Announcements

2 MST Review

3 Dijkstra's

4 A^*

Kruskal's Pseudocode

- 1 Initialize our output graph, $T = (V', E')$, $V' = V$, $E' = \{\}$
- 2 Initialize a disjoint sets structure S , where each vertex represents a set (all in own set to begin with)
- 3 Initialize a priority queue, P , holding all the **edges** in the original graph
- 4 $e = P.\text{removeMin}()$
 - If e connects two vertices from different sets, add e to E' , and union the two vertices from e in S
 - If e connects two vertices from the same set, do nothing (this would create a cycle)
- 5 Repeat step 4 until $|E'| = n - 1$

Kruskal's Analysis

Here's the outline of Kruskal's:

- Initialize disjoint sets: $O(m)$
- Initialize priority queue
- Call `removeMin` $n - 1$ times:
 - Call `union` if necessary: $\sim O(1)$

	Binary Heap	Sorted Array	AVL Tree
build each <code>removeMin</code>			

Kruskal's Analysis II

We call `build` once and `removeMin` n times. `DisjointSets` operations are effectively constant.

Using all this information, we can build the final running time for Kruskal's based on the following:

$$O(\text{build} + n \cdot \text{removeMin})$$

You should be able to plug in running times (and simplify) using any PQ structure. For Kruskal's, *the graph implementation doesn't matter!* This is assuming we start with all the edges. Otherwise, we need to do a traversal to get them. Don't forget:

$$n - 1 \leq m \leq n^2$$

Prim's Pseudocode

- 1 Initialize a priority queue P , to hold **vertices** based on a “cost” (initially all ∞); change a start vertex’s cost to 0
- 2 Initialize an empty dictionary D : vertex \rightarrow parent
- 3 $v = P.\text{removeMin}()$, label v as visited
 $\forall w \in \text{adjacent}(v)$:
 - if w is unvisited and $\text{cost}(v, w) < P[w]$
 - $P.\text{decreaseKey}(w, \text{cost}(v, w))$
 - $D[w] = v$
- 4 Repeat step 3 for n times (until P is empty), then create T by using the parents in D

Prim's Analysis

Here's an outline of Prim's (assuming D is a hash table):

- Initialize priority queue
- Call `removeMin` n times:
 - Call `getAdjacent`
 - Call `decreaseKey` if necessary

	Binary Heap	Fibonacci Heap
build		$O(n)$
each <code>removeMin</code>		$O(\log n)*$
each <code>decreaseKey</code>		$O(1)*$

But how many times do we call `decreaseKey`?

$$\sum_{v \in V} \deg(v) = 2m = O(m)$$

Prim's Analysis II

We call `build` once, `removeMin` n times, `getAdjacent` n times, and `decreaseKey` $2m$ times.

Using all this information, we can build the final running time for Prim's based on the following:

$$O(\text{build} + n \cdot (\text{removeMin} + \text{getAdjacent}) + m \cdot \text{decreaseKey})$$

You should be able to plug in running times (and simplify) using any PQ structure and any graph implementation! Also remember for connected graphs:

$$n - 1 \leq m \leq n^2$$

Outline

1 Announcements

2 MST Review

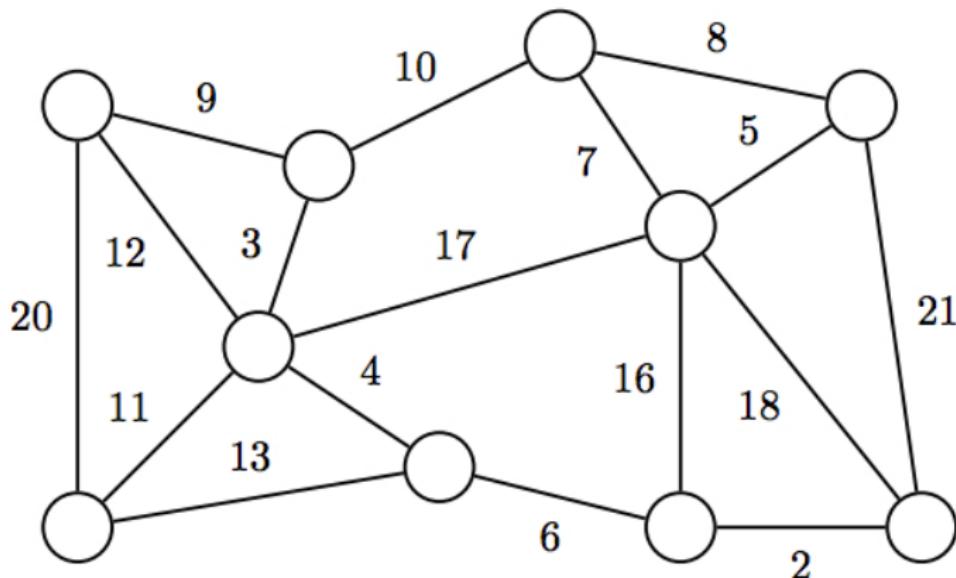
3 Dijkstra's

4 A^*

Single Source Shortest Path

- Input: directed graph with *non-negative edge weights*
- Output: a subgraph consisting of the shortest path (minimum total cost) from the start vertex to every other vertex in the graph
- Dijkstra's algorithm is a solution to this problem (1956)
- A* is another solution (1968), and is an extension to Dijkstra's algorithm

Example graph



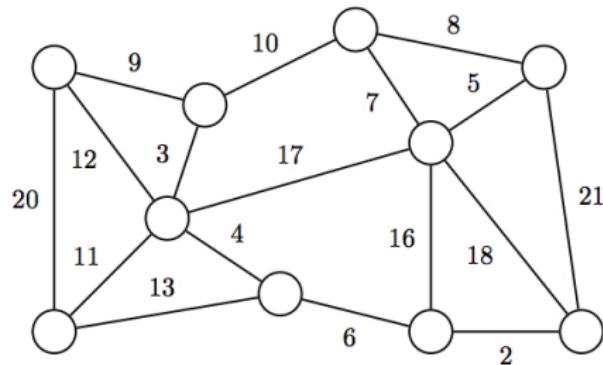
Dijkstra's Pseudocode

- 1 Initialize a priority queue P , to hold **vertices** based on a “cost” (initially all ∞); change a start vertex’s cost to 0
- 2 Initialize an empty dictionary D : vertex \rightarrow parent
- 3 $v = P.\text{removeMin}()$, label v as visited
 $\forall w \in \text{adjacent}(v)$:
 - if w is unvisited and $P[v] + \text{cost}(v, w) < P[w]$
 - $P.\text{decreaseKey}(w, P[v] + \text{cost}(v, w))$
 - $D[w] = v$
- 4 Repeat step 3 for n times (until P is empty), then create T by using the parents in D

It looks just like Prim's!

- 1 Initialize a priority queue P , to hold **vertices** based on a “cost” (initially all ∞); change a start vertex's cost to 0
- 2 Initialize an empty dictionary D : vertex \rightarrow parent
- 3 $v = P.\text{removeMin}()$, label v as visited
 $\forall w \in \text{adjacent}(v)$:
 - if w is unvisited and $\text{cost}(v, w) < P[w]$
 - $P.\text{decreaseKey}(w, \text{cost}(v, w))$
 - $D[w] = v$
- 4 Repeat step 3 for n times (until P is empty), then create T by using the parents in D

Running Dijkstra's



Dijkstra's Analysis: the same as Prim's

We call `build` once, `removeMin` n times, `getAdjacent` n times, and `decreaseKey` $2m$ times.

Using all this information, we can build the final running time for Dijkstra's based on the following:

$$O(\text{build} + n \cdot (\text{removeMin} + \text{getAdjacent}) + m \cdot \text{decreaseKey})$$

Of course, this just ends up being the same as Prim's analysis. You should know how to calculate the final running times for a variety of data structure combinations.

Outline

1 Announcements

2 MST Review

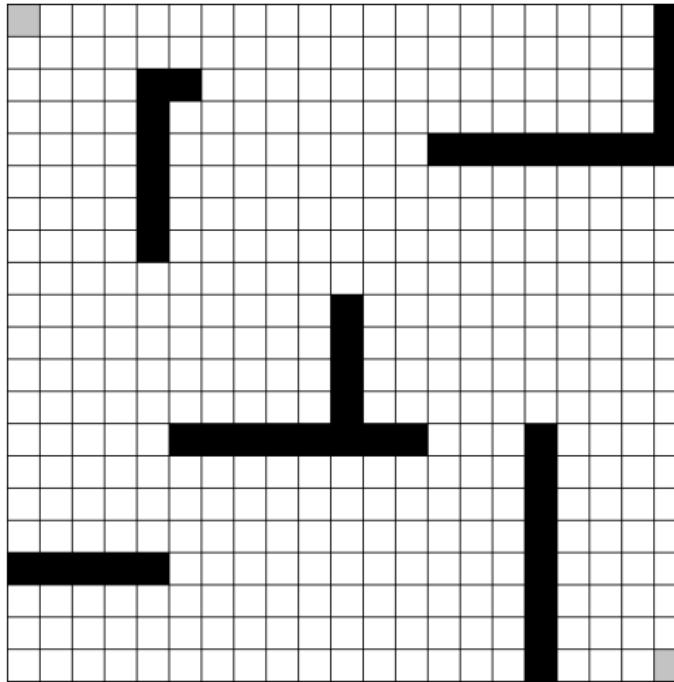
3 Dijkstra's

4 A^*

Searching for a particular node

- 1 Initialize a priority queue P , to hold vertices based on a “cost” (initially all ∞); change the start vertex’s cost to 0
- 2 Initialize an empty dictionary D : vertex \rightarrow parent
- 3 $v = P.\text{removeMin}()$, label v as visited
 - If v is the node we’re searching for, stop
 - Otherwise, $\forall w \in \text{adjacent}(v)$: if w is unvisited and $P[v] + \text{cost}(v, w) < P[w]$
 - $P.\text{decreaseKey}(w, P[v] + \text{cost}(v, w))$
 - $D[w] = v$
- 4 If P is empty, the destination was not found. Otherwise, go to step 3.

Searching on a grid



PathFinding.js

Compare BFS, Dijkstra's, and A^* :

<http://qiao.github.io/PathFinding.js/visual/>

Then some elevation map searches (for maps with non-uniform edge weights).

Heuristics

- A heuristic is an approximation or a guess
- In the case of A^* , a heuristic is used to guess how far from the a given node the destination is
- It's only possible to guess for certain types of graphs—specifically, graphs on grids give an easy way to guess a distance:
 - 1 Manhattan Distance
 - 2 Euclidean Distance
 - 3 Chebyshev Distance
- So, instead of only comparing the actual cost so far, a heuristic search algorithm also takes into account the estimated future cost

The heuristics

- Manhattan Distance

$$dist(p, q) = \sum_{i=0}^n |p_i - q_i|$$

- Euclidean Distance

$$dist(p, q) = \sqrt{\sum_{i=0}^n (p_i - q_i)^2}$$

- Chebyshev Distance

$$\max_i (|p_i - q_i|)$$

A* Outline

- Use a heuristic distance measurement to explore in the best-looking direction first
- Like Dijkstra's, store the cost so far at each node, but also include the estimated future cost in this number
- That way, assuming our heuristics are good, we can explore fewer nodes before we reach the target
- This is also known as a “best-first search”

A* Data Structures

- A priority queue P , to hold the actual + estimated costs to the goal
- A dictionary D_A , to map actual costs to explored nodes
- A dictionary D_P , to map nodes to parents (to reconstruct the path)
- A function $cost(v_i, v_j)$ just as before; these are the edge weights
- A function $h(v_i, v_k)$, which is one of the heuristic distance measures, so that v_i does not have to be adjacent to v_k to estimate the cost

A* Pseudocode

- 1 Initialize P , to hold vertices (initially all ∞)
- 2 $D_A[start] = 0, P.\text{decreaseKey}(start, D_A[start] + h(start, goal))$
- 3 $D_P = \{\}$
- 4 $v = P.\text{removeMin}()$. If $v = goal$, stop
- 5 $\forall w \in \text{adjacent}(v)$:
if $D_A[v] + \text{cost}(v, w) < D_A[w]$
 - $D_A[w] = D_A[v] + \text{cost}(v, w)$
 - $P.\text{decreaseKey}(w, D_A[w] + h(w, goal))$
 - $D[w] = v$
- 6 If P is empty, the destination was not found. Otherwise, go to step 4.

Related problems

- Dijkstra's is Prim's with a modified cost function
- BFS is Dijkstra's on graphs with identical edge weights; in this case, the priority queue turns into a FIFO queue
- Dijkstra's is A* without a heuristic
- A* is BFS with no heuristic and uniform edge weights

lecture30: Beyond CS 225

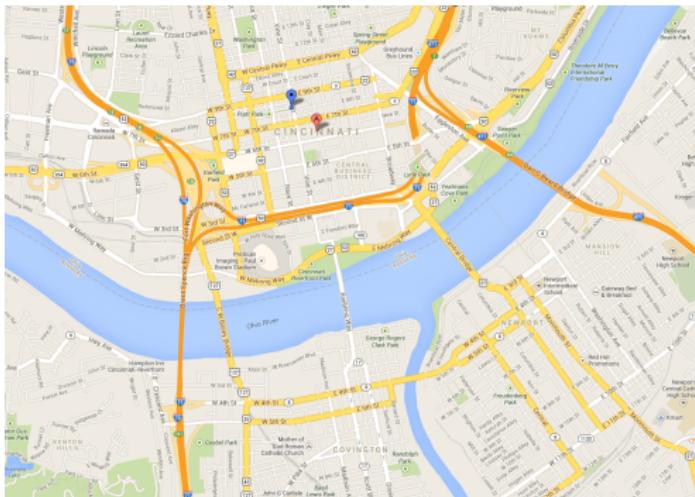
Sean Massung

CS 225 UIUC

31st July, 2013

Announcements

- lab_graphs due tomorrow, 8/1 at 11pm
- final exam this Friday (8/2), 10:30am-12:30pm in this room



Anything but “Hello World”

```
/** @file main.cpp */
#include <iostream>
using namespace std;

int main(int argc, char** args)
{
    int n = 47;
    cout << "My favorite number is " << n << endl;
    return 0;
}
```

- Compiles with g++ main.cpp -o myprog
- Execute with ./myprog
- What happens?

Three (really two) ways to pass variables

```
/**  
 * Passing by value  
 * makes a copy of  
 * the object  
 */
```

```
void kiwi(int a)  
{  
    a = 1;  
}
```

```
void main()  
{  
    int x = 0;  
    kiwi(x);  
}
```

```
/**  
 * You can also  
 * pass a pointer  
 * (by value) to  
 * the object  
 */
```

```
void pear(int* a)  
{  
    *a = 2;  
}
```

```
void main()  
{  
    int x = 0;  
    pear(&x);  
}
```

```
/**  
 * Passing by  
 * reference gives  
 * the variable a  
 * different name  
 */
```

```
void durian(int & a)  
{  
    a = 3;  
}
```

```
void main()  
{  
    int x = 0;  
    durian(x);  
}
```

Writing constructors

```
/** @file sphere.h */

#ifndef SPHERE_H
#define SPHERE_H

class Sphere
{
public:
    Sphere(double newRadius);
    void setRadius(double newRadius);

private:
    double radius;
};

#endif
```

```
/** @file sphere.cpp */

#include "sphere.h"

Sphere::Sphere(double newRadius)
{
    radius = newRadius;
}

Sphere::setRadius(double newRadius)
{
    radius = newRadius;
}
```

Don't duplicate your code!!

```
/** @file book.h */

class Book {

public:

    Book(const string & title);
    Book(const Book & other);
    Book & operator=(const Book & rhs);
    ~Book();

private:

    string _title;
    string* _lines;

    void copy(const Book & other);
    void clear();

};
```

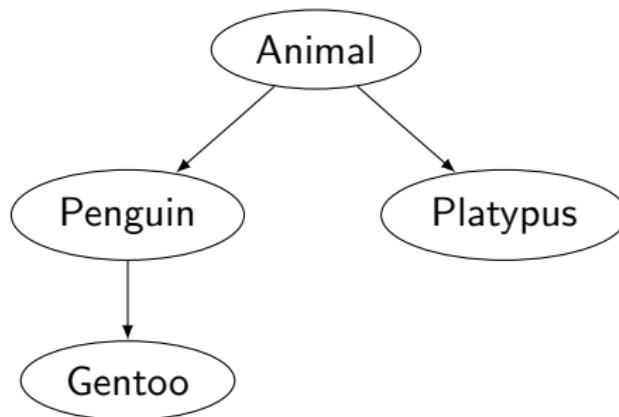
```
/** @file book.cpp */

Book::Book(const Book & other)
{
    copy(other);
}

Book & Book::operator=(const Book & rhs)
{
    if(this != &rhs) {
        clear();
        copy(rhs);
    }
    return *this;
}

Book::~Book()
{
    clear();
}
```

Inheritance diagram



*/** @file main.cpp */*

```
Penguin pen("Todd");  
pen.speak();  
pen.eat();
```

```
Platypus plat("Perry");  
plat.speak();  
plat.sleep();
```

```
Gentoo gent("Ted");  
gent.sleep();
```

Implementing a templated class

```
/** @file pair.h */

#ifndef PAIR_H
#define PAIR_H

template <class T>
class Pair
{
public:
    Pair(const T & first,
          const T & second);
    T max() const;
private:
    T _first;
    T _second;
};

#include "pair.cpp"
#endif
```

```
/** @file pair.cpp */
// no #include here

template <class T>
Pair<T>::Pair(const T & first,
               const T & second)
{
    _first = first;
    _second = second;
}

template <class T>
T Pair<T>::max() const
{
    if(_first > _second)
        return _first;
    return _second;
}
```

Our first data structure analysis

With all data structures in CS 225, we want to analyze the running times of the ADT functions. If n is the number of elements in the `Vector`, fill in the running times with Big- O notation:

Operation	Running time
<code>push_back</code>	
<code>pop_back</code>	
<code>at</code>	
<code>size</code>	
<code>empty</code>	

Writing insertFront

```
/** @file list.cpp */

template <class T>
List<T>::List(): head(NULL) { /* nothing */ }

template <class T>
List<T>::Node::Node(const T & newData):
    data(newData), next(NULL) { /* nothing */ }

template <class T>
void List<T>::insertFront(const T & data)
{
    // ok?
    head->next = new Node(data);
}
```

What choices do we have?

```
/** @file stack.h */

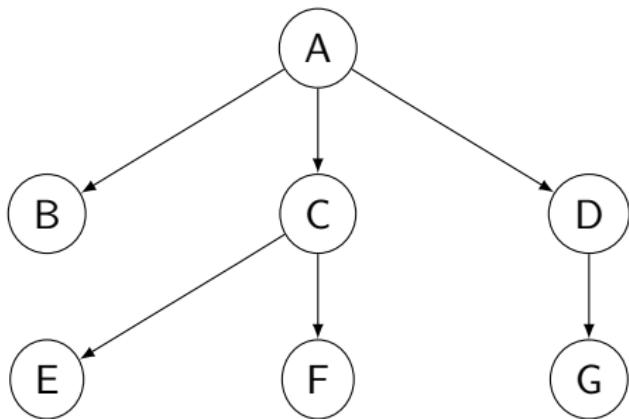
template <class T>
class Stack
{
public:
    Stack(); // + big 3
    void push(const T & elem);
    T pop();
    T peek() const;
    size_t size() const;
    bool empty() const;

private:
    // what should go in here?
};
```

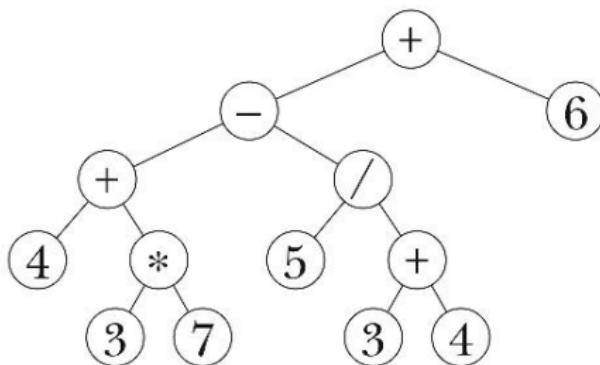
More pointers

In linked lists, each Node had one pointer to another Node. What kind of structures can we make if each Node has more than one pointer?

```
template <class T>
struct TreeNode
{
    T data;
    TreeNode* one;
    TreeNode* two;
    TreeNode* three;
    // etc...
};
```



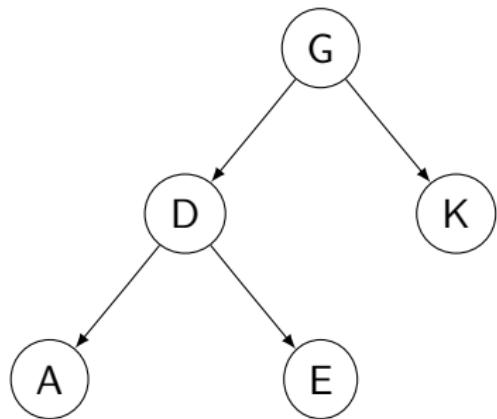
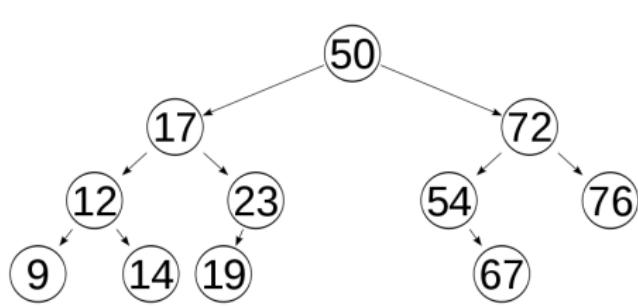
How should we visit each node?



We will discuss four kinds of traversals:

- 1 Pre-order
- 2 In-order
- 3 Post-order
- 4 Level order

A special kind of binary tree



A cute proof

Recall the Fibonacci numbers:

$F(n) = F(n - 1) + F(n - 2)$, $F(0) = 0$, $F(1) = 1$. What happens if we plot the values of the Fibonacci numbers along with $S(h)$?

h	0	1	2	3	4	5	6	7	8	...
$S(h)$	1	2	4	7	12	20	33	54	88	...
$F(h + 3)$	2	3	5	8	13	21	34	55	89	...

Do you see any connection?

$S(h) = F(h + 3) - 1$, could you prove it?

How would you implement a PriorityQueue?

Implementation options:

- 1 Unsorted or sorted array?
- 2 Unsorted or sorted linked list?
- 3 Balanced or unbalanced BST?

Remember, we need to support these PriorityQueue ADT functions:

- 1 push (any element)
- 2 pop (the highest priority element)
- 3 peek (show the next element that will be popped)

SUHA and hash table running times

We've mentioned previously that all dictionary ADT operations are constant (or amortized constant) time. Let's see how SUHA and table resizing can ensure this is true.

Operation	Worst Case	SUHA
insert	$O(1)^*$	$O(1)^*$
successful remove/find	$O(n)$	$O(\frac{1}{2}(\frac{n}{N})) = O(\frac{\alpha}{2}) = O(1)$
unsuccessful remove/find	$O(n)$	$O(\frac{n}{N}) = O(\alpha) = O(1)$

Recall we are using linked lists as our "buckets". Could we use some other data structure? How (if at all) would that change our running times?

We're modeling equivalence relations

What is an equivalence relation again?

A given binary relation \sim on a set S is said to be an equivalence relation if and only if it is reflexive, symmetric and transitive.

Equivalently, $\forall a, b, c \in S$:

- $a \sim a$ (reflexive)
- $a \sim b \rightarrow b \sim a$ (symmetric)
- $a \sim b \wedge b \sim c \rightarrow a \sim c$ (transitive)

How does this translate to our programming language example?

Our Graph ADT... our *last* ADT :'(

```
// assume we have a Vertex and Edge class, each templated
// to hold an arbitrary object

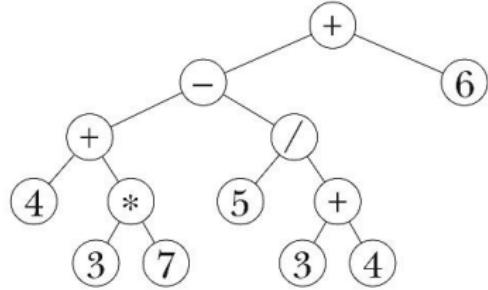
class Graph {
public:
    void insertVertex(const Vertex & v);
    void insertEdge(const Edge & e, Vertex v1, Vertex v2);
    void removeVertex(const Vertex & v);
    void removeEdge(const Edge & edge);

    vector<Edge> getIncident(const Vertex & v) const;
    vector<Vertex> getAdjacent(const Vertex & v) const;
    bool adjacent(const Vertex & v1, const Vertex & v2) const;

private:
    // ???
};
```

A level order traversal Breadth-First Search

```
template <class T>
void BinaryTree<T>::traverse() const
{
    Queue<TreeNode*> q;
    q.enqueue(root);
    while(!q.empty())
    {
        TreeNode* cur = q.dequeue();
        if(cur != NULL)
        {
            yell(cur->data);
            q.enqueue(cur->left);
            q.enqueue(cur->right);
        }
    }
}
```



First, explore all nodes at a distance one away, then at a distance two, then, three.....

Prim's Analysis II

We call `build` once, `removeMin` n times, `getAdjacent` n times, and `decreaseKey` $2m$ times.

Using all this information, we can build the final running time for Prim's based on the following:

$$O(\text{build} + n \cdot (\text{removeMin} + \text{getAdjacent}) + m \cdot \text{decreaseKey})$$

You should be able to plug in running times (and simplify) using any PQ structure and any graph implementation! Also remember for connected graphs:

$$n - 1 \leq m \leq n^2$$

Single Source Shortest Path

- Input: directed graph with *non-negative edge weights*
- Output: a subgraph consisting of the shortest path (minimum total cost) from the start vertex to every other vertex in the graph
- Dijkstra's algorithm is a solution to this problem (1956)
- A* is another solution (1968), and is an extension to Dijkstra's algorithm

Future Classes

- I'm a CS major, so I know about many other CS classes
- I'm sure there are many other interesting classes that have CS 225 as a prereq
- ...but I'll just list the CS courses that list CS 225 as a prerequisite

CS 241 (or ECE 391)

- “Basics of system programming, including POSIX processes, process control, inter-process communication, synchronization, signals, simple memory management, file I/O and directories, shell programming, socket network programming, RPC programming in distributed systems, basic security mechanisms, and standard tools for systems programming such as debugging tools.”
- Language: C
- Category: Systems

CS 373

- “Finite automata and regular languages; pushdown automata and context-free languages; Turing machines and recursively enumerable sets; computability and the halting problem; undecidable problems.”
- Language: none
- Category: Theory

CS 357 (MATH 357)

- “Fundamentals of numerical methods for students in science and engineering; floating-point computation, systems of linear equations, approximation of functions and integrals, the single nonlinear equation, and the numerical solution of ordinary differential equations; various applications in science and engineering; programming exercises and use of high quality mathematical library routines.”
- Language: MATLAB or Python (depending on instructor)
- Category: Numerical Analysis

CS 410

- “Theory, design, and implementation of text-based information systems. Text analysis, retrieval models (e.g., Boolean, vector space, probabilistic), text categorization, text filtering, clustering, retrieval system design and implementation, and applications to web information management.”
- Language: Your choice
- Category: Data Sciences

CS 411

- “Examination of the logical organization of databases: the entity-relationship model; the hierarchical, network, and relational data models and their languages. Functional dependencies and normal forms. Design, implementation, and optimization of query languages; security and integrity; concurrency control, and distributed database systems.”
- Language: Your choice
- Category: Data Sciences

CS 412

- “Concepts, techniques, and systems of data warehousing and data mining. Design and implementation of data warehouse and on-line analytical processing (OLAP) systems; data mining concepts, methods, systems, implementations, and applications.”
- Language: Your choice
- Category: Data Sciences

CS 440 (ECE 448)

- “Major topics in and directions of research in artificial intelligence: AI languages (LISP and PROLOG), basic problem solving techniques, knowledge representation and computer inference, machine learning, natural language understanding, computer vision, robotics, and societal impacts.”
- Language: Your choice
- Category: Machine Learning and AI

CS 473

- Ok, not a direct prereq for CS 225 (also need CS 373), but in many ways CS 473 is a continuation of CS 225
- “Fundamental techniques for algorithm design and analysis, including recursion, dynamic programming, randomization, dynamic data structures, fundamental graph algorithms, and NP-completeness.”
- Language: none
- Category: Theory

ICES Forms

Please let us know how we did!