坚持. Lecture slides ① ②

CS 225

#1: Introduction

January 17, 2018 · *Wade Fagen-Ulmschneider*

坚持到了学期末.

Welcome to CS 225: "Data Structures"

Course Website: https://courses.engr.illinois.edu/cs225/
...or just Google Search: `cs225 uiuc`

Description: Data abstractions: elementary data structures (lists, stacks, queues, and trees) and their implementation using an object-oriented programming language. Solutions to a variety of computational problems such as search on graphs and trees. Elementary analysis of algorithms.

Instructors:
Wade Fagen-Ulmschneider <waf@>, Teaching Asst. Prof.
Eric Shaffer <shaffer1@>, Teaching Asst. Prof.

Course Coordinator:
Thierry Ramais <ramais@>, Head of Course Logistics

Lab Sections: As part of registering for CS 225, you have also registered for a lab section. **Labs start this week** and are held by your contact TA – your lab TA is the first person who you should contact if you're having any troubles with the course!

Coursework and Grading

A total of 1,000 points are available in CS 225, along with many opportunities to earn extra credit (capped at +100 points). The points are broken down in the following way:

- **100 points**: Weekly lab assignments
  - There are 14 weeks -- points above 100 are extra credit!
- **260 points**: MPs (20 for MP1, 45 for MP2-7)
  - Early submissions of MPs earns up to +7 points /MP after MP1
- **490 points**: Bi-weekly Quizzes (CBTF)
  - "Quiz 0" (50 minutes, 40 points)
  - 3 "theory" quizzes (50 minutes, theory/math-focused)
  - 3 programming quizzes (110 minutes, coding focused)
- **150 points**: Final Exam (CBTF)
- Extra Credit: Problems of the Day (POTD)
  - Short, fun programming exercises to keep you coding!
  - +1 extra credit point /POTD, maximum of +40 points

Final Course Grades

Your final course grade is determined by the number of points you earned during the semester:

Points	Grade	Points	Grade	Points	Grade
[1070, ∞)	A+	[930, 1070)	A	[900, 930)	A-
[870, 900)	B+	[830, 870)	B	[800, 830)	B-
[770, 800)	C+	[730, 770)	C	[700, 730)	C-
[670, 700)	D+	[630, 670)	D	[600, 630)	D-
		(600, 0]	F		

We never curve individual exam or assignment scores; instead, if necessary, we may lower the points required for each grade cutoff to be lower than the stated cutoff. However, this has not been necessary in recent semesters. In no case will we raise the cutoff.

Contacting Us:

We want to do everything we can to make sure you have a great time learning data structures. There's a lot of ways to reach out if you get stuck:

1. **Course Piazza Site**: Online Q&A discussion forum to ask and answers from your peers and course staff. Join it!
   URL: http://piazza.com/illinois/spring2018/cs225

2. **Open Office Hours**: Course staff will be holding over 50 hours of office hours every week, starting next week. Great way to get help on MPs, labs, and technical content.
   Location: Basement of Siebel Center
   Times: Check CS 225's website "Calendar" for times

3. **Lab TA**: Your Lab TA is your primary administrative contact for the course. You will see them each week in lab and can reach them via e-mail. However, no coding assistance will be provided via e-mail (use open office hours).

4. **Debug Your Brain (DYB)**: Wade holds open office hours in ECEB after the 11am class and before the 2pm class. Come by and say hello, let me know how the course is going, or about any problems you're having. Not in a lab, not well suited for help on coding assignments (use open office hours for that).

Variables and Classes in C++

Every variable is defined by four properties:

1. type
2. name
3. value
4. memory location.

And every variable is one of two types:

`int myFavoriteInt;`
`char grade = 'A';`
`double gamma = 0.653;`
	`Sphere myFavoriteSphere;`
	`Cube rubix;`
	`Grade courseGrade;`

primitive	user-defined or complex

Creating New Types

In data structures, we will be learning and creating new types of structures to store data. We will start simply – by the end, we will have types we built being the building blocks for new types!

Big Idea: Encapsulation

Separation of the API from the implementation.

what it does

how it does it

Encapsulation principles:

sphere.h	sphere.cpp
1 Header file	1 cpp file
2 what my class	2 how the class is
3 does	3 implemented.
4	4
5	5

(3) (4). → inclusion guards.

Our first class:

sphere.h

```
1  #ifndef SPHERE_H_
2  #define SPHERE_H_
3
4  class Sphere {
5    public:
6      double getRadius();
7
8
9
10
11
12   private:
13     double radius_;
14
15 };
16 ☆ easy to miss out
17 #endif
```

sphere.cpp

```
1  #include "sphere.h"
2
3  double
     Sphere::getRadius() {
4      return radius_;
5
6  }
7
8
9
10
11
12
13
14
15
16
```

Key Concepts in C++ Classes:

Every class will be made up of common key pieces:

1. _____
   `sphere.h:1 :2 and :17`
2. _____
   `sphere.h:4..15`
3. _____
   `sphere.cpp:1`
4. _____
   `sphere.cpp:3`

CS 225 – Things To Be Doing:

1. Join the CS 225 Piazza
2. Visit our course website
3. Attend your lab sections (lab sections start **today!**)
4. MP1 is released Friday, due Monday after next (Jan. 29)

Lecture Slides (5) (6)

CS 225

#2: Classes and Reference Variables

January 19, 2018 · *Wade Fagen-Ulmschneider*

Our First Class – Sphere:

sphere.h

```
1  #ifndef SPHERE_H_
2  #define SPHERE_H_
3
4  class Sphere {
5    public:
6     double getRadius();
7
8     double getVol();
9
10
11   private:
12    double r_;
13
14 };
15
16 #endif
```

sphere.cpp

```
1  #include "sphere.h"
2
3  double
     Sphere::getRadius() {
4
5    return r_;
6  }
7
8  double Sphere :: getVol(){
9    return (4.0*3.14*r_*r_*r_)
10                /3.0;
11 }
12
13
14
15
```

Public vs. Private:

Situation	Protection Level
Helper function used internally in `Sphere`	private.
Variable containing data about the `Sphere`	private
`Sphere` functionality provided to client code	public

Hierarchy in C++:

There `Sphere` class we're building might not be the only `Sphere` class. Large libraries in C++ are organized into namespaces.

sphere.h

```
1  #ifndef SPHERE_H
2  #define SPHERE_H
3
4  namespace cs225 {
5    class Sphere {
6    public:
7      double getRadius();
…    /* ... */
```

sphere.cpp

```
1  #include "sphere.h"
2
3  namespace cs225 {
4    double
       Sphere::getRadius() {
5      return r_;
6    }
7  }
```

needed for both class declaration and constructor (namespace cs225).

Our first Program:

may cost too much for large namespaces.

main.cpp

```
1  #include "sphere.h"
2  #include <iostream>
3
4  int main() {
5    cs225::Sphere s;
6    std::cout << "Radius: " << s.getRadius() << std::endl;
7    return 0;
8  }
```

using namespace std;

…run this yourself: run `make main` and `./main` in the lecture source code.

Several things about C++ are revealed by our first program:

1. every c++ program starts at int main.
   `main.cpp:4`
2. able to access to "Sphere" class in cs225 namespace.
   `main.cpp:5, main.cpp:1`
3. able to access standard library
   `main.cpp:6, main:cpp:2`
4. However, our program is unreliable. **Why?**

the Sphere s is declared, but not constructed?

Default Constructor:

Every class in C++ has a constructor – even if you didn't define one!

- Automatic Default Constructor:
  ① every single class in C++ has a free default constructor. provided if we do not define a constructor.
- Custom Default Constructor:

sphere.h

```
…
4  class Sphere {
5    public:
6     Sphere();
…     /* ... */
```

sphere.cpp

```
…
3  Sphere::Sphere() {
4
5    r_ = 1.0;
6  }
…
```

② Initialize all variables to default values.
③ zero parameters.

Custom, Non-Default Constructors:
We can provide also create constructors that require parameters when initializing the variable:

sphere.h

```
4  class Sphere {
5    public:
6     Sphere(double r);
     /* ... */
```

sphere.cpp

```
3  Sphere::Sphere(double r) {
4
5    r_ = r;
6  }
```

Puzzle #1: How do we fix our first program?

main.cpp w/ above custom constructor

```
8   Sphere s;
9   cout << "Radius: " << s.getRadius() << endl;
```

use a default constructor (0-parameter)

...run this yourself: run `make puzzle` and `./puzzle` in the lecture source code.

compiler error.

Solution #1:
write in a customer default constructor (0-parameter).

Solution #2:
call the customer non-default constructor
`Sphere S(7);`

The beauty of programming is both solutions work! There's no one right answer, both have advantages and disadvantages!

Pointers and References – Introduction
A major component of C++ that will be used throughout all of CS 225 is the use of references and pointers. References and pointers both:

- Are extremely power, but extremely dangerous
- Are a **level of indirection** via memory to the data.

As a level of indirection via memory to the data:

1. create an alias to existing data.
2. multiple variables may modify the same memory

Often, we will have direct access to our object:

```
Sphere s1;  // A variable of type Sphere
```

Occasionally, we have a reference or pointer to our data:

```
Sphere & s1;  // A reference variable of type Sphere
Sphere * s1;  // A pointer that points to a Sphere
```

Reference Variable
A reference variable is an alias to an existing variable. Modifying the reference variable modifies the variable being aliased. Internally, a reference variable maps to the same memory as the variable being aliased:

main-ref.cpp

```
3  int main() {
4    int i = 7;
5    int & j = i;   // j is an alias of i
6
7    j = 4;                              // j and i are both 4.
8    std::cout << i << " " << j << std::endl;
9
10   i = 2;                              // j and i are both 2.
11   std::cout << i << " " << j << std::endl;
12   return 0;
13 }
```

...run this yourself: run `make main-ref` and `./main-ref` in the lecture source code.

Three things to note about reference variables:

1. always contain a reference to data (cannot be "NULL").
2. Never creates new memory (aliasing somebody else's memory)

int &a = 3; ✗ error!

3. Reference variables are defined when initialized and then can never be changed.

CS 225 – Things To Be Doing:

1. Sign up for "Exam 0" (starts Tuesday, Jan. 23rd)
2. Complete lab_intro; due Sunday, Jan. 21st
3. MP1 released today; due Monday, Jan. 29th
4. Visit Piazza and the course website often!

Lecture Slides. TS.

CS 225

#3: Memory

January 22, 2018 · *Wade Fagen-Ulmschneider*

Pointers and References

Often, we will have direct access to our object:

```
Sphere s1;  // A variable of type Sphere
```

Occasionally, we have a reference or pointer to our data:

```
Sphere & s1;  // A reference variable of type Sphere
Sphere * s1;  // A pointer that points to a Sphere
```

Pointers

Unlike reference variables, which alias another variable's memory, pointers are variables with their own memory. Pointers store the memory address of the contents they're "pointing to".

Three things to remember on pointers:

1. pointers store a memory address. - not the object/data itself.
2. May point to nothing. → NULL (memory address 0.)
3. powerful, but dangerous!

main.cpp

```
4  int main() {
5    cs225::Sphere s;
6    std::cout << "Address storing `s`:" << &s << std::endl;
7
8    cs225::Sphere *ptr = &s;
9    std::cout << "Addr. storing ptr: "<< &ptr << std::endl;
10   std::cout << "Contents of ptr: "<< ptr << std::endl;
11
12   return 0;
13 }
```

~~get the content stored in s~~, the memory address of sphere's ?

declaration. 0x ffff 0000.

the content of "ptr", the address of c.

~~the memory address storing~~ ptr.

Indirection Operators:

*(&v) ≡ v.

&v returns the memory address storing v.

*v returns the data at the address stored in v.

v-> Accesses a member of an object pointer

(*v).

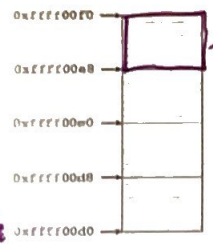
```
Sphere *s;
s->getRadius();
(*s).getRadius();
```


Stack Memory:

① Starts at high addresses and grows toward zero.

② By default, all memory is created on the stack.

③ allocate and then write our data.



example1.cpp

```
1  int main() {
2    int a;
3    int b = -3;
4    int c = 12345;
5
6    int *p = &b;
7
8    return 0;
9  }
```

Location	Value	Type	Name
0xffff00f0 (start)	?	int	a
0xffff00e8	-3	int	b
0xffff00e0	12345	int	c
0xffff00d8	0xffff00e8	int *	p
0xffff00d0			

sizeof(int) depends on system & compiler.

clang++ on ews

int → 4 bytes.

address of b.

0xffff00e8

+....

read backward.

example2.cpp

```
3  int main() {
4    cs225::Sphere s;
5    cs225::Sphere *p = &s;
6
7    return 0;
8  }
```

Location	Value	Type	Name
0xffff00f0			
0xffff00e8			
0xffff00e0		Sphere	s
0xffff00d8	0xffff00e0	* Sphere	p.
0xffff00d0			

sizeof(s) → 8

sizeof(~~*p~~) → 8

p.

Stack Frames

All variables (including parameters to the function) that are part of a function are part of that function's **stack frame**. A stack frame:

1. created when a function is called.
2. memory is freed when a function returns.

stackframe.cpp

```
1  int hello() {
2    int a = 100;
3    return a;
4  }
5
```

```
6  int main() {
7    int a;
8    int b = -3;
9    int c = hello();
10   int d = 42;
11
12   return 0;
13 }
```

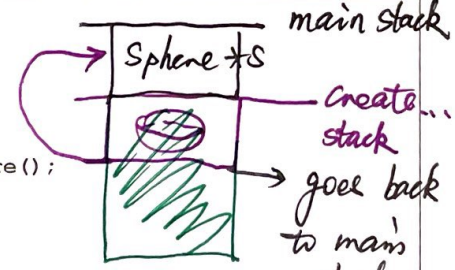
Location	Value	Type	Name
0xffff00f0 →	a	main's stack	
0xffff00e8 →	b = -3		
0xffff00e0 →	c = 100		
0xffff00d8 →	d	hello()'s stack	
0xffff00d0 →	a = 100		

this piece of memory was used before. what's in it?

Puzzle: What happens here?

puzzle.cpp

```
4  Sphere *CreateUnitSphere() {
5    Sphere s(1);
6    return &s;
7  }
8
9  int main() {
10   Sphere *s = CreateUnitSphere();
11   double r = s->getRadius();
12   double v = s->getVolume();
13   return 0;
14 }
```



these two lines of code may put gabbage and contaminate Sphere s.

Heap Memory:

As programmers, we can use heap memory in cases where the lifecycle of the variable exceeds the lifecycle of the function.

1. The only way to create heap memory is with the use of the **new** keyword. Using **new** will:
   -
   -
   -
2. The only way to free heap memory is with the use of the **delete** keyword. Using **delete** will:
   -
   -
3. Memory is never automatically reclaimed, even if it goes out of scope. Any memory lost, but not freed, is considered to be "leaked memory".

heap1.cpp

```
4  int main() {
5    int *p = new int;
6    int *s = new Sphere(10);
7
8    return 0;
9  }
```

Stack	Value
0xffff00f0 →	
0xffff00e8 →	
0xffff00e0 →	
0xffff00d8 →	
0xffff00d0 →	

Heap	Value
0x42020 →	
0x42018 →	
0x42010 →	
0x42008 →	
0x42000 →	

CS 225: TTBD → Exam 0 starts tomorrow; MP1 due Monday (Jan. 29)

CS 225

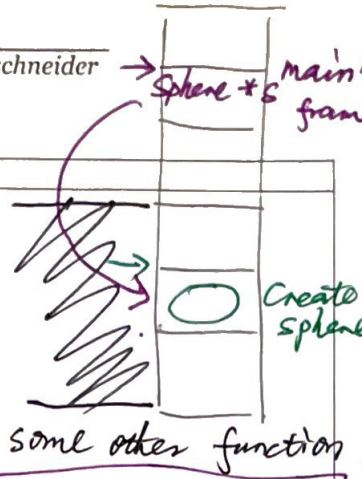
#4: Heap Memory

January 24, 2018 · *Wade Fagen-Ulmschneider*

Puzzle from Monday

puzzle.cpp

```
4  Sphere *CreateUnitSphere() {
5    Sphere s(1);
6    return &s;
7  }
8
9  int main() {
10   Sphere *s = CreateUnitSphere();
11   someOtherFunction();
12   double r = s->getRadius();
13   double v = s->getVolume();
14   return 0;
15 }
```



Never return a ref to stack memory.

Takeaway:

"where is the variable stored?"

"either on stack or on heap".

Second type of memory.

Heap Memory:

As programmers, we can use heap memory in cases where the *lifecycle of the variable exceeds the lifecycle of the function.*

grows from small addresses toward ∞.

1. The only way to create heap memory is with the use of the **new** keyword. Using **new** will:
   - allocate heap memory
   - call object constructor if necessary.
   - return a pointer to the memory (points to the starting of the memory (lowest address))

   Last entire life cycle of the program.
2. The only way to free heap memory is with the use of the **delete** keyword. Using **delete** will:
   - call the objects destructor
   - mark the heap memory as free.

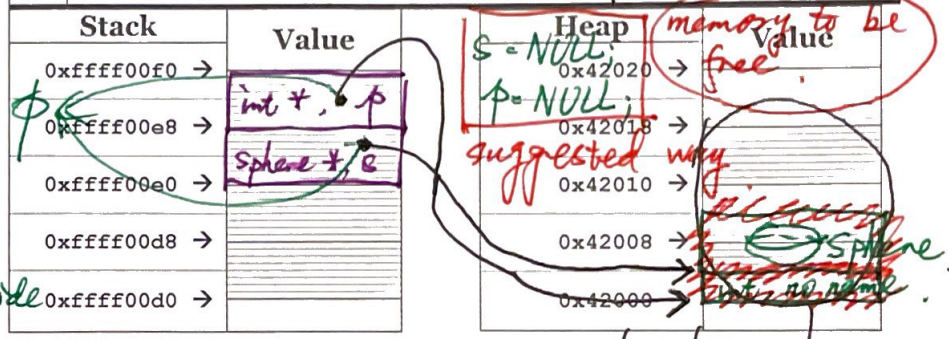
3. Memory is never automatically reclaimed, even if it goes out of scope. Any memory lost, but not freed, is considered to be "leaked memory".

pointers are 8 bytes.

heap1.cpp

```
4  int main() {
5    int *p = new int;
6    Sphere *s = new Sphere(10);
7    s->getVolume();
8    return 0;
9  }
```

delete s; remove the "box" on the heap.
delete p;

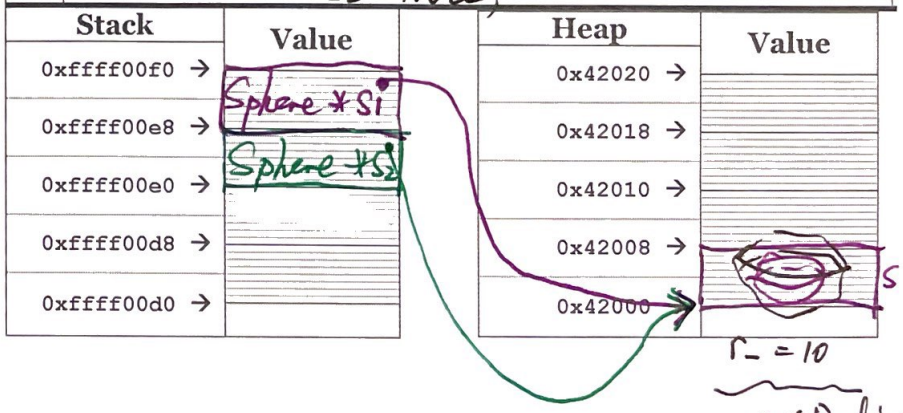


heap2.cpp

```
4  int main() {
5    Sphere *s1 = new Sphere();
6    Sphere *s2 = s1;
7    s2->setRadius( 10 );
8    delete s2;
9    delete s1;
10   return 0;
11 }
```

don't do that!

s1 = NULL;
s2 = NULL;



Copying Memory – Deep Copy vs. Shallow Copy

copy.cpp

```
5  int i = 2, j = 4, k = 8;
6  int *p = &i, *q = &j, *r = &k;
7
8  k = i;
9  cout << i << j << k << *p << *q << *r << endl;
10
11 p = q;
12 cout << i << j << k << *p << *q << *r << endl;
13
14 *q = *r;
15 cout << i << j << k << *p << *q << *r << endl;
```

Deep copy

Shallow copy

Consider how each assignment operator changes the data:

	Type of LHS	Type of RHS	Data Changed?
Line 8-9			
	i =	j =	k =
	p =	q =	r =
Line 11-12			
	i =	j =	k =
	p =	q =	r =
Line 14-15			
	i =	j =	k =
	p =	q =	r =

heap-puzzle1.cpp

```
5  int *x = new int;
6  int &y = *x;
7
8  y = 4;
9
10 cout << &x << endl;
11 cout << x << endl;
12 cout << *x << endl;
13
14 cout << &y << endl;
15 cout << y << endl;
16 cout << *y << endl;
```


heap-puzzle2.cpp

```
5  int *p, *q;
6  p = new int;
7  q = p;
8  *q = 8;
9  cout << *p << endl;
10
11 q = new int;
12 *q = 9;
13 cout << *p << endl;
14 cout << *q << endl;
```

heap-puzzle3.cpp

```
5  int *x;
6  int size = 3;
7
8  x = new int[size];
9
10 for (int i = 0; i < size; i++) {
11   x[i] = i + 3;
12 }
13
14 delete[] x;
```

joinSpheres.cpp

```
11 /*
12  * Creates a new sphere that contains the exact volume
13  * of the two input spheres.
14  */
15 Sphere joinSpheres(Sphere   s1, Sphere   s2) {
16   double totalVolume = s1.getVolume() + s2.getVolume();
17
18   double newRadius = std::pow(
19     (3.0 * totalVolume) / (4.0 * 3.141592654),
20     1.0/3.0
21   );
22
23   Sphere result(newRadius);
24
25   return  result;
26 }
```

CS 225 – Things To Be Doing:

1. Exam 0 is ongoing – ensure you're signed up for it!
2. Finish up MP1 – Due Monday, Jan. 29 at 11:59pm
3. Complete lab_debug this week in lab sections (due Sunday)
4. POTDs are released daily, worth +1 extra credit point! ☺

CS 225

#5: Parameters

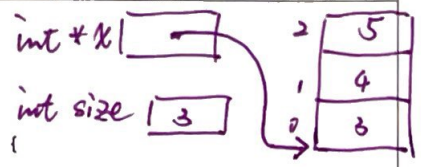
January 26, 2018 · *Wade Fagen-Ulmschneider*

Heap Memory – Allocating Arrays

heap-puzzle3.cpp

```
5  int *x;
6  int size = 3;
7
8  x = new int[size];
9
10 for (int i = 0; i < size; i++) {
11   x[i] = i + 3;
12 }
13
14 delete[] x;
```

Stack / Heap



8 bytes on stack.

delete for array, special []

*: **new[]** and **delete[]** are identical to **new** and **delete**, except the constructor/destructor are called on each object in the array.

Memory and Function Calls

Suppose we want to join two Spheres together:

joinSpheres-byValue.cpp

```
11 /*
12  * Creates a new sphere that contains the exact volume
13  * of the sum of volume of two input spheres.
14  */
15 Sphere joinSpheres(Sphere s1, Sphere s2) {
16   double totalVolume = s1.getVolume() + s2.getVolume();
17
18   double newRadius = std::pow(
19     (3.0 * totalVolume) / (4.0 * 3.141592654),
20     1.0/3.0
21   );
22
23   Sphere result(newRadius);
24
25   return result;
26 }
```

By default, arguments are "passed by value" to a function. This means that:

- Data is copied to the function stack frame.
- Modifying don't change the original data.

no copies are made.

Alterative #1: Pass by Reference

but aliasing the memory

joinSpheres-byReference.cpp

```
15 Sphere joinSpheres(Sphere & s1, Sphere & s2) {
16   double totalVolume = s1.getVolume() + s2.getVolume();
17
18   double newRadius = std::pow(
19     (3.0 * totalVolume) / (4.0 * 3.141592654),
20     1.0/3.0
21   );
22
23   Sphere result(newRadius);
24
25   return result;
26 }
```

modifications do change the original data.

Alternative #2: Pass by Pointer

joinSpheres-byPointer.cpp

```
15 Sphere joinSpheres(Sphere * s1, Sphere * s2) {
16   double totalVolume = s1->getVolume() + s2->getVolume();
17
18   double newRadius = std::pow(
19     (3.0 * totalVolume) / (4.0 * 3.141592654),
20     1.0/3.0
21   );
22
23   Sphere result(newRadius);
24
25   return result;
26 }
```

	By Value	By Pointer	By Reference
Exactly what is copied when the function is invoked?	entire object (may be large)	1. pointer (8 Bytes)	Nothing only aliasing the memory.
Does modification of the passed in object modify the caller's object?	No.	Yes.	Yes. looks facinate.
Is there always a valid object passed in to the function?	Yes.	No.	Yes. refs have to be bonded to sth.
Speed	Dependent on data, maybe slow	Fast & constant	Fast & constant
Safety	safe.	Not safe.	use care.

Using the const keyword

1. Using const in function parameters:

Make a promise, not to modify the variable.

joinSpheres-byValue-const.cpp

```
15  Sphere joinSpheres(const Sphere  s1, const Sphere  s2)
15  Sphere joinSpheres(const Sphere *s1, const Sphere *s2)
15  Sphere joinSpheres(const Sphere &s1, const Sphere &s2)
```

Best Practice: *"All parameters passed by reference must be labeled const."* – *Google C++ Style Guide*

consistent of declaration and calling.

2. Using const as part of a member functions' declaration:

Make a promise not to modify the state of a class.

sphere-const.h

```
 5  class Sphere {
 6    public:
 7      Sphere();
 8      Sphere(double r);
 9
10      double getRadius() const;
11      double getVolume() const;
12
13      void setRadius(double r);
14
...     // ...
```

sphere-const.cpp

```
...
15  double Sphere::getRadius() const {
16    return r_;
17  }
18
19  double Sphere::getVolume() const {
20    return (4 * 3.14 * r_ * r_ * r_) / 3.0;
21  }
...
```

Returning from a function

Identical to passing into a function, we also have three choices on how memory is used when returning from a function:

Return by value:

```
15  Sphere  joinSpheres(const Sphere &s1, const Sphere &s2)
```

Return by reference:

```
15  Sphere &joinSpheres(const Sphere &s1, const Sphere &s2)
```

...remember: never return a reference to stack memory!

Return by pointer:

```
15  Sphere *joinSpheres(const Sphere &s1, const Sphere &s2)
```

...remember: never return a reference to stack memory!

Copy Constructor

When a non-primitive variable is passed/returned **by value,** a copy must be made. As with a constructor, an automatic copy constructor is provided for you if you choose not to define one:

All **copy constructors** will:

The **automatic copy constructor**:

1.

2.

To define a **custom copy constructor**:

sphere.h

```
 5  class Sphere {
 6    public:
 7      Sphere(const Sphere & other);   // custom copy ctor
```

CS 225 – Things To Be Doing:

1. Theory Exam #1 begins Tuesday – ensure you're registered!
2. lab_debug due Sunday (11:59pm)
3. mp1 due Monday (11:59pm)
4. Complete POTDs

CS 225

#6: Lifecycle of Classes

January 29, 2018 · *Wade Fagen-Ulmschneider*

Returning from a function

Identical to passing into a function, we also have three choices on how memory is used when returning from a function:

Return by value:

```
15  Sphere  joinSpheres(const Sphere &s1, const Sphere &s2)
```

Return by reference:

```
15  Sphere &joinSpheres(const Sphere &s1, const Sphere &s2)
```

...remember: never return a reference to stack memory!

Return by pointer:

```
15  Sphere *joinSpheres(const Sphere &s1, const Sphere &s2)
```

...remember: never return a reference to stack memory!

Copy Constructor

When a non-primitive variable is passed/returned **by value,** a copy must be made. As with a constructor, an automatic copy constructor is provided for you if you choose not to define one:

All **copy constructors** will: build an object from another existing object of the same type.

The **automatic copy constructor**:

1. only exists if no copy constructor is defined.
2. makes a copy of all member variables.

To define a **custom copy constructor**:

sphere.h

```
5   class Sphere {
6     public:
7       Sphere();           // default ctor
8       Sphere(double r);   // 1-param ctor
9       Sphere(const Sphere & other);  // custom copy ctor
10      ...
```

sphere.cpp. specify the same type of the class.

```
Sphere::Sphere(const Sphere & other) {
  r_ = other.r_;
```


Bringing Concepts Together:

How many times do our different joinSphere files call each constructor?

	By Value	By Pointer	By Reference
`Sphere()`	0 times	0	0
`Sphere(double)`	1 times	1	1
`Sphere(const Sphere &)`	(2+1) times	0	0

* ← pass pointer

joinSpheres-{byValue,byReference,byPointer}.cpp

```
15  Sphere joinSpheres(Sphere  X  s1, Sphere  X  s2) {   X ← pass value
16    double totalVolume = s1.getVolume() + s2.getVolume();
17                         →                →          copy constructor
18    double newRadius = std::pow(
19      (3.0 * totalVolume) / (4.0 * 3.141592654),
20      1.0/3.0
21    );
22
23    Sphere result(newRadius);   1-para constructor
24
25    return result;   return by value, copy
26  }  return new Sphere(newRadius);   return by pointer
                                       no copy.
```

A Sphere, A Universe.

Consider a Universe of three Spheres:

Universe.h

```
1   #ifndef UNIVERSE_H_
2   #deifne UNIVERSE_H_
3
4   #include "Sphere.h"
5   using namespace cs225;
6
7   class Universe {
8     public:
9       Universe();         // default ctor
10      Universe(Sphere s, Sphere *q, Sphere &r);  // 3-param
11      Universe(const Universe & other);
12      // …
13    private:
14      Sphere p_, *q_, &r;
15  };
16        OK, but looks bad.
17  #endif
```

New Class

p q r

23

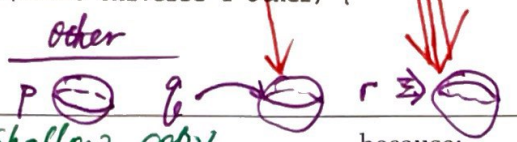
Automatic Copy Constructor Behavior:

The behavior of the automatic copy constructor is to make a copy of every variable. We can mimic this behavior in our Universe class:

Universe.cpp

```
10 Universe::Universe(const Universe & other) {
11   p_ = other.p_;
12   q_ = other.q_;
13   r_ = other.r_;
14 }
```

other



...we refer to this as a Shallow copy because:

Universe::Universe(const Universe & other):

pointers are copied, not the object.

p_(other.p_), q_(other.q_), r_(other.r_) { }

Deep Copy via Custom Copy Constructor:

Alternatively, a custom copy constructor can perform a deep copy:

Universe.h

```
16 Universe::Universe(const Universe & other) {
17   // Deep copy p_:
18
19     p_ = other.p_;
20
21   // Deep copy q_:                    copy constructor by reference?
22
23     q_ = new Sphere(*(other.q_));    Always!
24
25   // Deep copy r_:
26
27     You can't.  references do not have
28
29 }   shouldn't.  own memory.
```

Destructor

The last and final member function called in the lifecycle of a class is the destructor.

Purpose of a **destructor**:

The **automatic destructor**:

1.

2.

26

Custom Destructor:

sphere.h

```
 5 class Sphere {
 6   public:
 7     Sphere();            // default ctor
 8     Sphere(double r);    // 1-param ctor
 9     Sphere(const Sphere & other);   // custom copy ctor
10     ~Sphere();           // destructor, or dtor
11     ...
```

Overloading Operators

C++ allows custom behaviors to be defined on over 20 operators:

Arithmetic	+ - * / % ++ --
Bitwise	& | ^ ~ << >>
Assignment	=
Comparison	== != > < >= <=
Logical	! && ||
Other	[] () ->

General Syntax:

Adding overloaded operators to Sphere:

sphere.h

```
 1 #ifndef SPHERE_H
 2 #define SPHERE_H
 3
 4 class Sphere {
 5   public:
 …     // ...
17
18
19
20
 …     // ...
```

sphere.cpp

```
 …  /* ... */
10
11
12
13
14
15
16
17
18
 …  /* ... */
```

CS 225 – Things To Be Doing:

1. Theory Exam #1 Starts Tomorrow (Register in the CBTF)
2. MP1 due tonight; grace period until Tuesday @ 11:59pm
3. MP2 released on Tuesday *(start early for extra credit!)*
4. Lab Extra Credit → Attendance in your registered lab section!
5. Daily POTDs every M-F for daily extra credit!

CS 225

#7: Inheritance

January 31, 2018 · *Wade Fagen-Ulmschneider*

Destructor

The last and final member function called in the lifecycle of a class is the destructor.

Purpose of a **destructor**: Free any resources stored by the class.

The **automatic destructor**:

1. Like a constructor and copy constructor, an automatic destructor exists only when no custom destructor is defined.
2. [Invoked]: never called explicitly. stack – out of scope; heap – deleted.
3. [Functionality]: call all member destructors.

Custom Destructor:

sphere.h

```
5  class Sphere {
6    public:
7      Sphere();            // custom default ctor
8      Sphere(double r);    // 1-param ctor
9      Sphere(const Sphere & other);   // custom copy ctor
10     ~Sphere();           // destructor, or dtor
11     ...
```

...necessary if you need to delete any heap memory!

Overloading Operators

C++ allows custom behaviors to be defined on over 20 operators:

Arithmetic	+ - * / % ++ --
Bitwise	& | ^ ~ << >>
Assignment	=
Comparison	== != > < >= <=
Logical	! && ||
Other	[] () ->

General Syntax: _____ operator + (RHS).

e.g. const Sphere & s.

Adding overloaded operators to Sphere:

sphere.h

```
1  #ifndef SPHERE_H
2  #define SPHERE_H
3
4  class Sphere {
5    public:
...    // ...
17
18
19
20
...  // ...
```

sphere.cpp

```
...  /* ... */
10
11
12   果然是年纪大了.
13   有吗? 谢谢!
14
15
16
17
18
...  /* ... */
```

One Very Powerful Operator: Assignment Operator

sphere.h

```
Sphere & operator=(const Sphere & other);
```

sphere.cpp

```
Sphere & Sphere::operator=(const Sphere & other) { ... }
```

Functionality Table:

	Copies an object	Destroys an object
Copy constructor	✓	
Assignment operator	② Replace with copied state.	① Destroy the current state.
Destructor		✓

The Rule of Three

If it is necessary to define any one of these three functions in a class, it will be necessary to define all three of these functions:

1. Copy constructor
2. Assignment operator
3. Destructor.

*Base → Base / Derived &Base → Base / Derived. (27) (28) polymorphed object ?

Inheritance

In nearly all object-oriented languages (including C++), classes can be extended to build other classes. We call the class being extended the **base class** and the class inheriting the functionality the **derived class**.

Derived Class: `Planet`

Planet.h

```
1  #ifndef PLANET_H_
2  #define PLANET_H_
3
4  #include "Sphere.h"
5
6  class Planet : public cs225::Sphere {
7     // Empty!
8  };
9
10 #endif
```

Sphere — Base

Planet — Derived

In the above code, `Planet` is derived from the base class `Sphere`:

- All public functionality of `Sphere` is part of `Planet`:

planet-main.cpp

```
5  int main() {
6    Planet p;
7    p.getRadius(); // Returns 1, the radius init'd
8                   // by Sphere's default ctor
…    ...
   }
```

- [Private Members of `Sphere`]:

p.r_ (Error!)

Cannot be accessed.

Adding New Functionality:

Planet.h

```
6  class Planet : public cs225::Sphere {
7    public:
8      Planet(std::string name, double radius); // ctor
9    private:
10     std::string name_;
11 };
```

If not specified, then default constructor for Sphere is invoked.

```
Planet::Planet(std::string name, double radius) : Sphere(radius) {
  name_ = name;
}
```

Functions: non-virtual, virtual, and pure virtual

- The **virtual** keyword: Allows for a class method to be replaced by a derived class.

Sphere.cpp

```
Sphere::print_1() {
  cout << "Sphere" << endl;
}

Sphere::print_2() {
  cout << "Sphere" << endl;
}

virtual Sphere::print_3() {
  cout << "Sphere" << endl;
}

virtual Sphere::print_4() {
  cout << "Sphere" << endl;
}

// .h:
virtual Sphere::print_5() = 0;
```

Planet.cpp

```
// No print_1() defined
// in Planet


Planet::print_2() {
  cout << "Earth" << endl;
}

// No print_3() defined
// in Planet


Planet::print_4() {
  cout << "Earth" << endl;
}

Planet::print_5() {
  cout << "Earth" << endl;
}
```

"pure virtual function" when a virtual function = 0. ← must be overwritten by a derived class

	`Sphere obj;`	`Planet obj;`	`Planet r; Sphere &obj = r;`
`obj.print_1();`	Sphere	Sphere	Sphere
`obj.print_2();`	Sphere	Earth	Sphere
`obj.print_3();`	Sphere.	Sphere (inherited from "Sphere.h")	Sphere.
`obj.print_4();`	Sphere	Earth.	Earth (because print_4 is virtual)
`obj.print_5();`	cannot create a class w/ a pure virtual (not compile)	Earth.	Earth.

"obj" is a sphere.

first check if virtual. if not, does not look at derived class. if yes, replace with derived

CS 225 – Things To Be Doing:

1. Theory Exam #1 is ongoing – ensure you're signed up!
2. Attend and complete lab_memory *(due Sunday)*
3. MP2 is ongoing *(extra credit due Monday)*
4. Daily POTDs every M-F for daily extra credit!

CS 225

#8: Templates

February 2, 2018 · *Wade Fagen-Ulmschneider*

Polymorphism

Object-Orientated Programming (OOP) concept that a single object may take on the type of any of its base types.

- A Planet may polymorph itself to a Sphere
- A Sphere cannot polymorph to be a Planet *(base types only)*

Virtual

- The **virtual** keyword allows us to override the behavior of a class by its derived type.

Sphere.cpp

```
Sphere::print_1() {
  cout << "Sphere" << endl;
}

Sphere::print_2() {
  cout << "Sphere" << endl;
}

virtual Sphere::print_3() {
  cout << "Sphere" << endl;
}

virtual Sphere::print_4() {
  cout << "Sphere" << endl;
}

// .h:
virtual Sphere::print_5() = 0;
```

Planet.cpp

```
// No print_1() defined
// in Planet


Planet::print_2() {
  cout << "Earth" << endl;
}

// No print_3() defined
// in Planet


Planet::print_4() {
  cout << "Earth" << endl;
}

Planet::print_5() {
  cout << "Earth" << endl;
}
```

	Sphere obj;	Planet obj;	Planet r; Sphere &obj = r;
obj.print_1();			
obj.print_2();			
obj.print_3();			
obj.print_4();			
obj.print_5();			

Why?

Suppose you're managing an animal shelter that adopts cats and dogs.

Option 1 – No Inheritance

animalShelter.cpp

```
1  Cat & AnimalShelter::adopt() { ... }
2  Dog & AnimalShelter::adopt() { ... }
3  ...
```

Option 2 – Inheritance

animalShelter.cpp

```
1  Animal & AnimalShelter::adopt() { ... }
```

Pure Virtual Methods

In `Sphere`, `print_5()` is a **pure virtual** method:

Sphere.h

```
1  virtual Sphere::print_5() = 0;
```

A pure virtual method does not have a definition and makes the class and **abstract class**.

Abstract Class:

1. [Requirement]: Any class with at least one pure virtual function.
2. [Syntax]: Nothing else. this is enough.
3. [As a result]: No instance of an abstract class can be created.

Abstract Class Animal

In our animal shelter, `Animal` is an abstract class:

Assignment Operator

I didn't cover a few details of the assignment operator -- let's do that:

1. [Default Assignment Operator]

C++ generate a default assignment operator for simple classes:

- No non-static const variables
- No reference variables

★ : Destructors shall always be virtual if the class might be a base class

2. [Self-Assignment]

- Programmers are never perfect and are never optimal. Consider the following:

assignmentOpSelf.cpp

```
1  #include "Sphere.h"
2
3  int main() {
4    cs225::Sphere s(10);
5    s = s;
6    return 0;
7  }
```

we don't want "segmentation fault" here.

- Ensure your assignment operator doesn't self-destroy:

assignmentOpSelf.cpp

```
1  #include "Sphere.h"
2
3  Sphere& Sphere::operator=(const Sphere &other) {
4    if (&other != this) {
5      _destroy();
6      _copy(other);
7    }
     return *this;
   }
```

★ : ensures not self-destroying

Abstract Data Types (ADT):

List ADT - Purpose	Function Definition

List Implementation

What types of List do we want?

Templated Functions:

functionTemplate1.cpp

```
1  template <typename T>
2
3  T maximum(T a, T b) {
4    T result;
5    result = (a > b) ? a : b;
6    return result;
7  }
```

Templated Classes:

List.h

```
1  #ifndef LIST_H
2  #define LIST_H
3
4
5
6  class List {
7    public:
8
9
10
11   private:
12
13
14 };
15
16 #endif
```

List.cpp

```
1
2
3
4
5
```

CS 225 – Things To Be Doing:

1. Theory Exam #1 – Today's the final day of the exam.
2. MP2 due Jan. 12 (10 days), EC deadline in 3 days!
3. Lab Extra Credit → Attendance in your registered lab section!
4. Daily POTDs

CS 225

#9: Templates and Linked Memory

February 5, 2018 · *Wade Fagen-Ulmschneider*

Abstract Data Types (ADT): Basic functionality of a data structure.

List ADT - Purpose	Function Definition
add / insert	store a piece of data.
remove	remove data.
get	access an element of data
size / is empty	how much data is stored.

create — create an empty list.

List Implementation

What types of List do we want? must be of same type.

generic across all kinds of data.

Templated Functions: allow us to write data structures with generic types.

functionTemplate1.cpp

```
1
2  template <typename T>
3  T maximum(T a, T b) {
4    T result;
5    result = (a > b) ? a : b;
6    return result;
7  }
```

int ✓ double ✓

cs225::Sphere ✗

The type used in the function ("<") is not defined for Sphere.

Templated Classes:

List.h

```
1  #ifndef LIST_H
2  #define LIST_H
3
4
5  template <typename T>
6  class List {
7    public:
8      T & get(unsigned index) const;
9
10
```



```
11   private:
12
13
14 };
15
16 #endif
```

List.cpp

```
1  template <typename T>
2  T & List::get(unsigned index) const {
3
4
5  }
```

should be in front of every function using T

return by reference.

Two Basic Implementations:

1. Array
2. Linked List (Memory)

Linked Memory:

① T data

② ListNode * next

ListNode

List.h

```
28 class ListNode {
29   T & data;
30   ListNode * next;
31   ListNode(T & data) : data(data), next(NULL) { }
32 };
```

private: → public:

head

Coding with Linked Lists: Examples

35

List.h

```
1  #ifndef LIST_H
2  #define LIST_H
3
4  template <typename T>
5  class List {
6    public:
...     /* ... */
28   private:
29     class ListNode {
30       T & data;
31       ListNode * next;
32       ListNode(T & data) : data(data), next(NULL) { }
33     };
34
35
36     ListNode * head_;
37
38
39
40 };
41
42 #endif
```

= class

originally heritated from C standard. Now change "public:" to "typename"

a "public" inside "private"

a constructor.

call the constructor

List.cpp

```
1  #include "List.h"
2
3  template <typename T>
4  void List::insertAtFront(T & t) {
5
6    ListNode * e_ = new ListNode(t);
7
8    head_
9
10   e_ -> next = head_;
11
12 }
```

head_ = e_;

running time

O(1)

or constant time.

36

List.cpp

```
14 void List::printReverse() const {
15
16
17
18
19
20
21
22 }
```

List.cpp

```
24 template <typename T>
25 T List::operator[](unsigned index) {
26
27
28
29
30
31 }
32
33 ListNode *& List::_index(unsigned index) {
34
35
36
37
38
39 }
```

CS 225 – Things To Be Doing:

1. Programming Exam A starts Feb. 13 (a week from tomorrow)
2. MP2 due Feb. 12 (7 days), EC deadline is tonight!
3. Lab Extra Credit → Attendance in your registered lab section!
4. Daily POTDs

CS 225

#10: List Implementations

February 7, 2018 · *Wade Fagen-Ulmschneider*

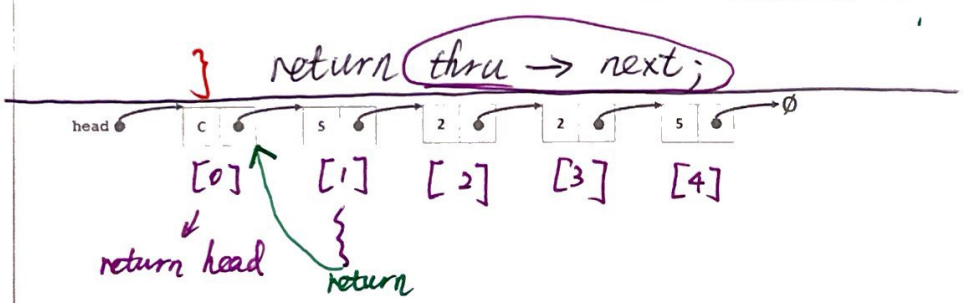
Finding in a list:

```
List.cpp
 1  #include "List.h"
 2
 3  ListNode *& List::_find(unsigned index) const {
 4
 5      if (index == 0) { return head; }
 6      ListNode * thru = head;
 7      for (unsigned i=0; i<(index-1); i++){
 8
 9
10
11
12  }       thru = thru -> next;
```

return a pointer by reference (a reference to a pointer)

what about NULL here?

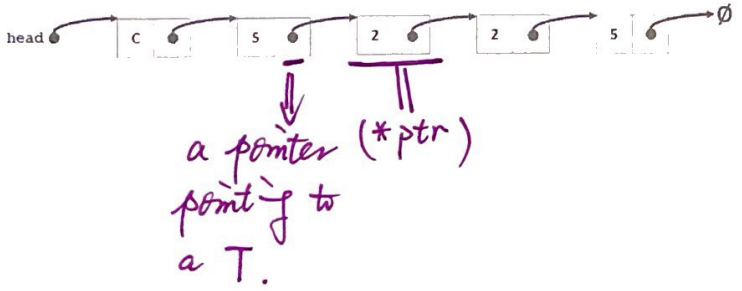
What is the return type of `_find`?



Building functionality with _find():

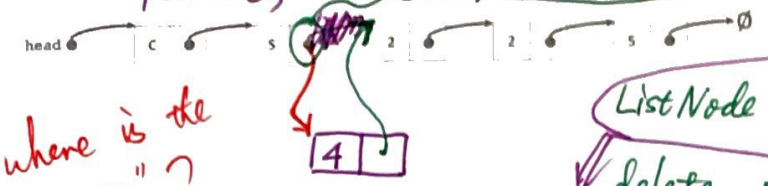
```
List.cpp
11  T & List::get(unsigned index) const {
12
13      ListNode *& ptr = _find(index);
14
15      return ptr -> data;
16  }
```

O(n).




```
List.cpp - insert()
11  T & List::insert(T & t, unsigned index) {
12      ListNode *& ptr = _find(index);
13
14      ListNode *e = new ListNode(t);
15
16  }   e -> next = ptr;
        ptr = e;
```

returned pointer



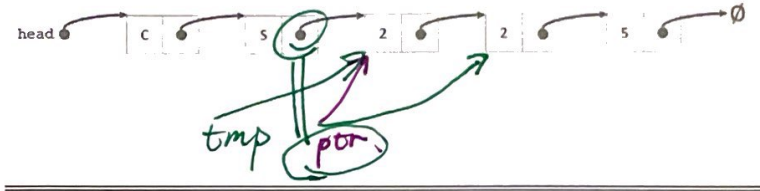
where is the "return"?

ListNode *temp = ptr;

delete ptr;

```
List.cpp - remove()        Not correct !
11  T & List::remove(unsigned index) {
12      ListNode *& ptr = _find(index);
13
14      ptr = ptr -> next;
15
16  }   // check end of list.
        // delete removed ListNode.
```

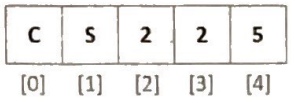
delete temp;



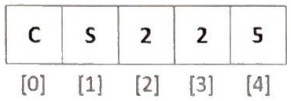
List Implementation #2: _____

```
List.h
 1  #ifndef LIST_H
 2  #define LIST_H
 3
 4  template <class T>
 5  class List {
 6    public:
...       /* ... */
28    private:
29
30
31
32  };
33
34  #endif
```

Array - Implementation Details:

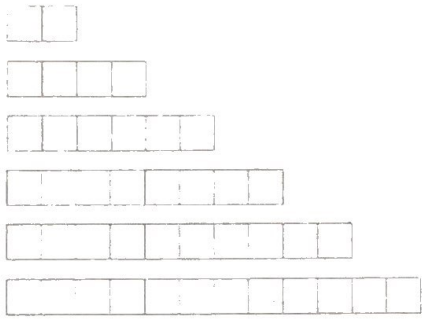


1. What is the running time of `insertFront()`?

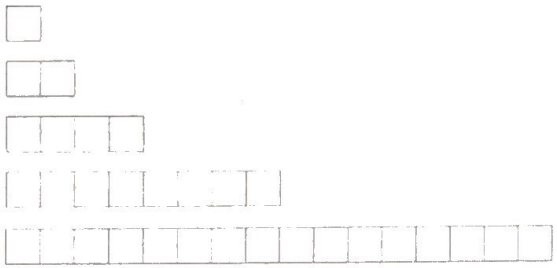


2. What is our resize strategy?

Resize Strategy #1:

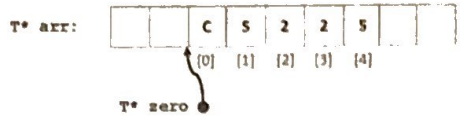


Resize Strategy #2:



39 40

3. What is the running time of get()?



	Singly Linked List	Array
Insert/Remove at **front**		
Insert after a **given** element		
Remove after a **given** element		
Insert at **arbitrary** location		
Remove at **arbitrary** location		

Stack ADT

Function Name	Purpose

CS 225 – Things To Be Doing:

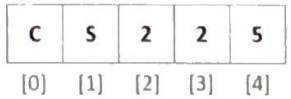
1. Programming Exam A starts Feb. 13 *(6 days from today)*
2. MP2 due Feb. 12 *(5 days from now)*, earn extra credit starting early!
3. Lab Extra Credit → Attendance in your registered lab section!
4. Daily POTDs

CS 225

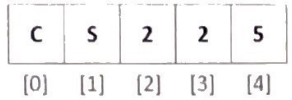
#11: Stacks and Queues

February 9, 2018 · *Wade Fagen-Ulmschneider*

Array-backed List - Implementation Details:

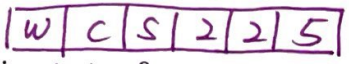


1. What is the running time of `insertFront()`?



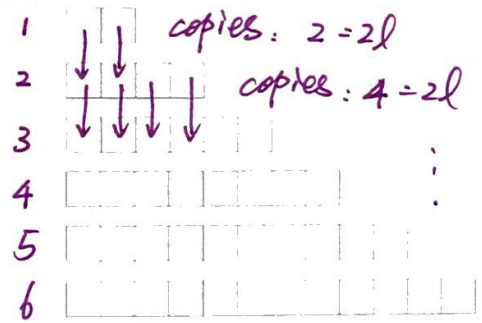
O(n) time.

↓ copy n elements



2. What is our resize strategy?

Resize Strategy #1:



Total copies: $\sum_{k=0}^{l} 2l = O(n^2)$.

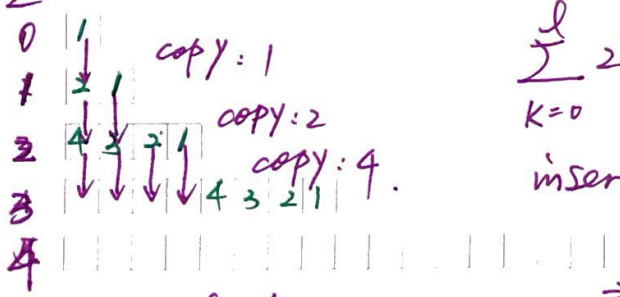
insert n elements.

⇒ O(n) work per insert.

l=6, n=12.

n=2l.

~~Resize Strategy #2:~~



$\sum_{k=0}^{l} 2^k = 2^{l+1} - 1 = O(n)$

inserts n : $\frac{O(n)}{n} = O(1)$

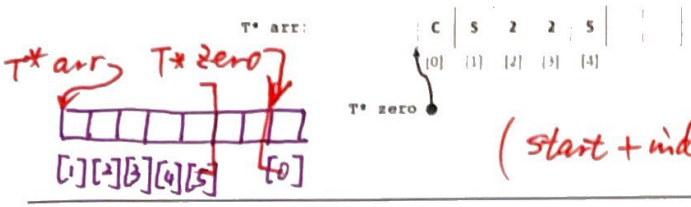
$l = \log_2 n$

⇒ O(1) work per insert.

(41)

(42) return arr_[(zero_ - arr_) + index % capacity_];

3. What is the running time of get()? O(1)



(start + index) % capacity

	Singly Linked List	Array
Insert/Remove at **front** both are happy	O(1)	O(1)*
Insert after a **given** element	O(1)	O(n)
Remove after a **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)+O(1)=O(n)	O(1)+O(n)=O(n)
Remove at **arbitrary** location	O(n)	O(n).
Access an index	O(n)	O(1).

Do a tradeoff.

Stack ADT

First in, Last out.

Function Name	Purpose
push	add an element
pop	remove an element
isEmpty	
create	

Queue ADT

First in, First out.

Function Name	Purpose
enqueue	
dequeue	
isEmpty	
create.	

$= \frac{1-2^{l+1}}{1-2} = \frac{2^{l+1}-1}{2-1}$

Stack and Queue Implementations

Stack.h

```
1  #ifndef STACK_H_
2  #define STACK_H_
3
4  #include "List.h"
5
6  template <typename T>
7  class Stack {
8    public:
9      void push(T & t);
10     T & pop();
11     bool isEmpty();
12
13   private:
14     List<T> list;
15
16 };
17
18 #endif
```

Stack.cpp

```
1  #include "Stack.h"
2
3  template <typename T>
4  void Stack::push(T & t) {
5    list_.add(t, 0);
6  }
7
8  template <typename T>
9  T & Stack::pop() {
10   return list_.remove(0);
11 }
12
13 bool Stack::isEmpty() {
14   return list_.isEmpty();
15 }
```

Three designs for data storage in data structures:

1. T & data

2. T * data

3. T data

Implication of Design

1. Who manages the lifecycle of the data?

2. Is it possible to store a NULL as the data?

3. If the data is manipulated by user code while stored in our data structure, are the changes reflected within our data structure?

4. Speed

	Storage by Reference	Storage by Pointer	Storage by Value
Lifecycle management of data?			
Possible to insert NULL?			
External data manipulation?			
Speed			

CS 225 – Things To Be Doing:

1. Programming Exam A starts Feb. 13 *(next Tuesday)*
2. MP2 due Feb. 12 *(next Monday)*
3. lab_inheritance due Sunday
4. Daily POTDs

CS 225

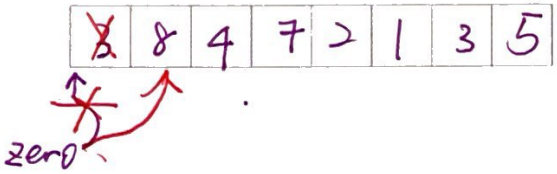
#12: Iterators

February 12, 2018 · *Wade Fagen-Ulmschneider*

Data Structures Review

- List ADT
  - Linked Memory Implementation ("Linked List")
    - O(1) insert/remove at front/back
    - O(1) insert/remove after a given element
    - O(n) lookup by index
  - Array Implementation ("Array List")
    - O(1) insert/remove at front/back
    - O(n) insert/remove at any other location
    - O(1) lookup by index
- Queue ADT
  - FIFO: First in, first out – *like a line/queue at a shop*
  - Implemented with a list, O(1) enqueue/dequeue
- Stack ADT
  - LIFO: Last in, first out – *list a stack of papers*
  - Implemented with a list, O(1) push/pop

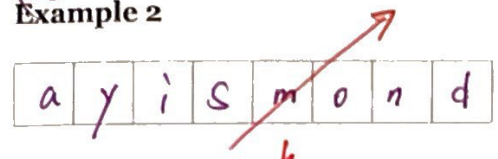
Example 1




```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

circular array?

Example 2



expand. double the size.

On day is h

must keep the sequence correctly.

```
Queue<char> q;
q.enqueue('m');
q.enqueue('o');
q.enqueue('n');
...
q.enqueue('d');
q.enqueue('a');
q.enqueue('y');
q.enqueue('i');
q.enqueue('s');
q.dequeue();
q.enqueue('h');
q.enqueue('a');
```

Three designs for data storage in data structures:

1. T & data
2. T * data
3. T data

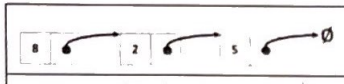
Tradeoffs between our data store strategies:

1. Who manages the lifecycle of the data?
2. Is it possible to store a NULL as the data?
3. If the data is manipulated by user code while stored in our data structure, are the changes reflected within our data structure?
4. What is the relative speed compared to other methods?

	Storage by Reference	Storage by Pointer	Storage by Value
Lifecycle management of data?	client code	client code	we have a copy.
Possible to insert NULL?	No NULL	Yes	No !
External data manipulation?	Yes	Yes	No.
Speed	Fast	Fast.	Slow.

Accessing Every Element in Our List / Queue / Stack

Suppose we want to look through every element in our data structure. What if we don't know what our data structure even looks like?

	Linked List
	Array
	Hypercube

Iterators

In C++, iterators provide an interface for client code access to data in a way that abstracts away the internals of the data structure.

An instance of an iterator is a current location in a pass through the data structure:

Type	Cur. Location	Current Data	Next
Linked List	ListNode ptr	curr → data	curr → next
Array	index	data [index]	index ++
Hypercube	(x, y, z) coordinate ?		

The iterator minimally implements three member functions:

- `operator*`, Returns the current data
- `operator++`, Advance to the next data
- `operator!=`, Determines if the iterator is at a different location

Implementing an Iterator

A class that implements an iterator must have two pieces:

1. [Implementing Class]:
   :: start (): starting point of the iteration.
   :: end (): end point of the iteration.
2. [Implementing Class' Iterator]:
   A separate class (usually an internal public member class) that extends `std::iterator` and implements an iterator.

Using an Iterator

stlList.cpp

```
1  #include <list>
2  #include <string>
3  #include <iostream>
4
5  struct Animal {
6    std::string name, food;
7    bool big;
8    Animal(std::string name = "blob", std::string food = "you",
   bool big = true) :
9      name(name), food(food), big(big) { /* none */ }
10 }
11
12 int main() {
13   Animal g("giraffe", "leaves", true),
           p("penguin", "fish", false), b("bear");
14   std::list<Animal> zoo;
15
16   zoo.push_back(g);
17   zoo.push_back(p);   // std::list's insertAtEnd
18   zoo.push_back(b);
19
20   for ( std::list<Animal>::iterator it = zoo.begin();
                                        it != zoo.end(); it++ )
21 {
22     std::cout << (*it).name << " " << (*it).food << std::endl;
23   }
24
25   return 0;
}
```

Seems like a class, while everything is public.

1. N, F, B.
2. N.
3. N, F
4. _ _ _

4 constructors.

Q: What does the above code do?

For-Each loop with Iterators

stlList-forEach.cpp

```
20 for ( const Animal & animal : zoo ) {
21   std::cout << animal.name << " " << animal.food << std::endl
22 }
```

CS 225 – Things To Be Doing:

1. Programming Exam A starts Feb. 13 *(tomorrow!)*
2. MP2 due tonight; MP3 released tomorrow
3. lab_quacks released on Wednesday
4. Daily POTDs

我如何与老师做一些不有脑子的事情 ?!

CS 225

#13: Trees

February 14, 2018 · *Wade Fagen-Ulmschneider*

(49)

Iterator Design: 我从来没听懂过哥们的英语!

[Monday's Lecture]: To implement an iterator, the implementing class must have two member functions:

- ::start(), returns an iterator at the first element
- ::end(), returns an iterator one past the end

why iterators are important ?

Queue.h

```
4  template <class QE>
5  class Queue {
6    public:
7      class QueueIterator :
       public std::iterator<std::forward_iterator_tag, QE> {
8        public:
9          QueueIterator(unsigned index);
10         QueueIterator& operator++();
11         bool operator==(const QueueIterator &other);
12         bool operator!=(const QueueIterator &other);
13         QE& operator*();
14         QE* operator->();
15       private:
16         int location_;
17
18     };
19   QueueIterator <QE> begin();
20
21   QueueIterator <QE> end();
22   /* ... */
23   private:
24     QE* arr_; unsigned capacity_, count_, entry_, exit_;
25 };
```

derived from base: std iterator

How does the `Queue` and the `QueueIterator` interact?

Two big takeaways:

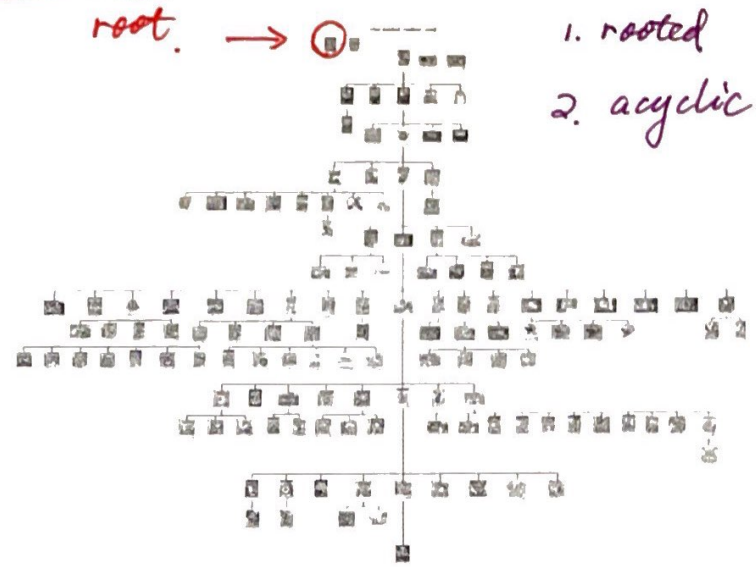
1.

2.

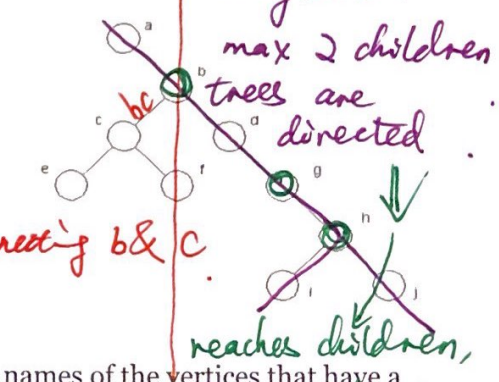
(50)

Trees!

"The most important non-linear data structure in computer science."
- David Knuth, The Art of Programming, Vol. 1



- We will primarily talk about **binary trees**
- What's the longest "word" you can make using the **vertex** labels in the tree (repeats allowed)?
- Find an **edge** that is not on the longest **path** in the tree. Give that edge a reasonable name.
- One of the vertices is called the **root** of the tree. Which one?
- Make a "word" containing the names of the vertices that have a **parent** but no **sibling**.
- How many parents does each vertex have?
- Which vertex has the fewest **children**?
- Which vertex has the most **ancestors**?
- Which vertex has the most **descendants**?
- List all the vertices is b's left **subtree**.
- List all the **leaves** in the tree.



node ← vertex

acyclic.
max 2 children
trees are directed.
bc
reaches children.

"bc" → the edge connecting b & c.

a

"bgh". I exapt root cannot go to parent (vertex ≠ root).

i, j, e, f.

i, j

a

c, e, f.

i, j, e, f.

Definition: Binary Tree

A *binary tree* **T** is either:

$T = \{ d, T_L, T_R \}$

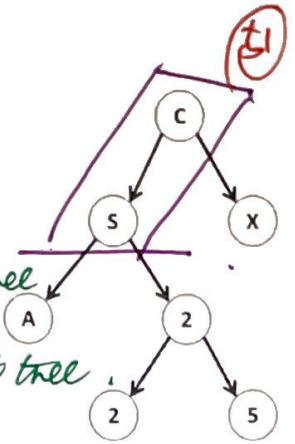
or

$T = \{\} = \phi$

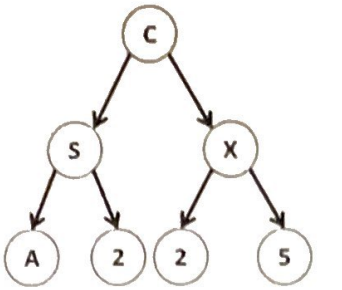
d: data

T_L: left tree

T_R: right tree



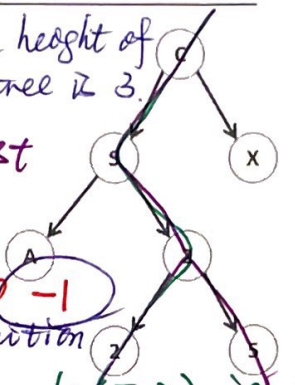
Tree Property: Perfect



The tree in the purple box is

$T = \{ "C", \{ "S", \phi, \phi \}, \phi \}$

Tree Property: Tree Height



The height of this tree is 3.

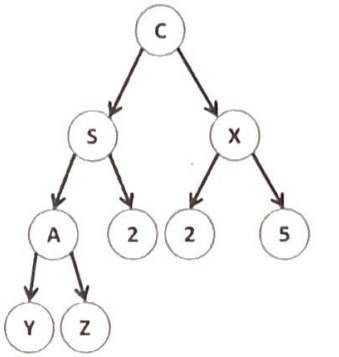
height(T) = Length of the longest path from the root to leaf.

height(ϕ) = 0. height($\{\}$) = ? −1 by definition

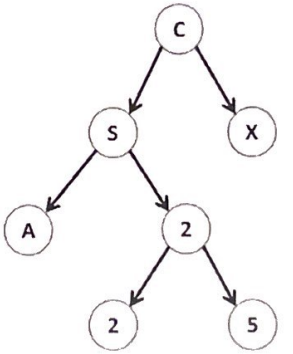
for binary tree T:

$$height(T) = \begin{cases} 1 + \max(height(T_L), height(T_R)) & \text{if } T \neq \{\} \\ -1, & \text{if } T = \{\} \end{cases}$$

Tree Property: Complete



Tree Property: Full



CS 225 – Things To Be Doing:

1. Programming Exam A is ongoing
2. MP3 has been released; extra credit deadline is Monday!
3. lab_quacks in lab this week
4. Daily POTDs

CS 225

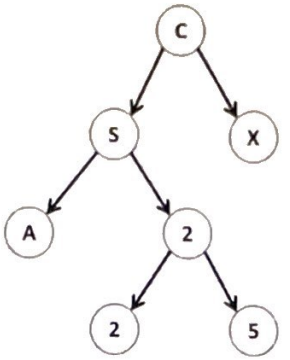
#14: Tree Proof – NULLs in a Tree

February 16, 2018 · *Wade Fagen-Ulmschneider*

Definition: Binary Tree

A *binary tree* **T** is:
$T = \{ d, T_L, T_R \}$ *or* $T = \{\}$

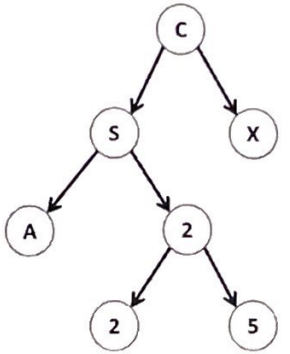
The height of a tree **T** is:
If $T = \{\}$, height(T) = -1
Otherwise:
height(T) = 1 + max(height(T_L), height(T_R))



Tree Property: Full

1. $F = \{ \}$
2. $F = \{ data, T_L, T_R \}$

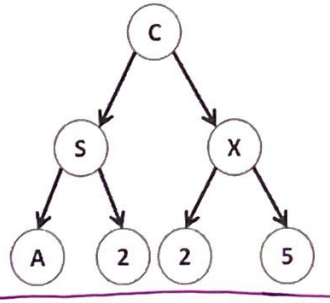
where T_L and T_R are
both FULL or
both EMPTY



Tree Property: Perfect

Let P_h be a perfect tree of height h. and.

1. $P_{-1} = \{ \} = \phi$
2. $P_h = \{ data, P_{h-1}, P_{h-1} \}$

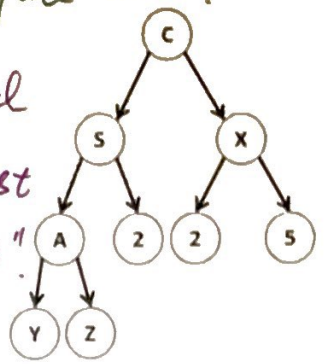


P_0	P_1	P_2
0		

Perfect tree ⊆ Complete tree.

Tree Property: Complete

A perfect tree for every level except the last, where the last level is "pushed to the left".



For all levels k in [0, h-1], k ~~has~~ 2^k nodes. For level h, all nodes are "pushed to the left"

Tree Class

BinaryTree.h

```
1  #ifndef BINARYTREE_H_
2  #define BINARYTREE_H_
3
4  template <typename T>
5  class BinaryTree {
6    public:
7
8      /* ... */
9
10   private:
11
12   struct TreeNode {
13
14     TreeNode * left;
15     TreeNode * right;
16     T& data;
17   };
18   #endif TreeNode(T& t)
```

Recursive definition:

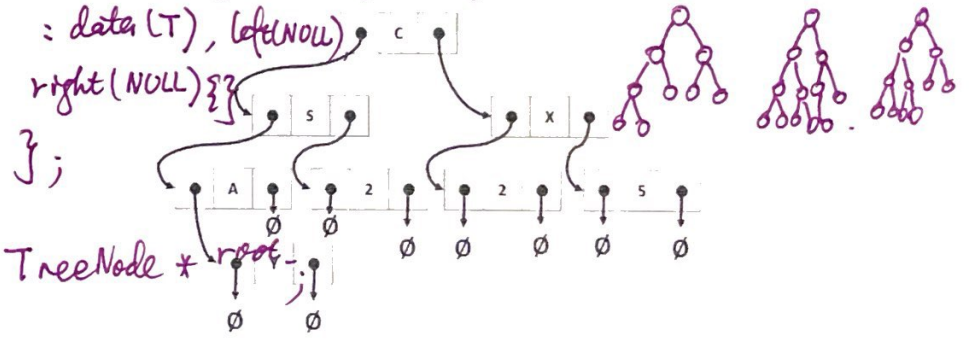
A Complete tree of height h, C_h is

1. $C_{-1} = \{ \}$
2. C_h (h>0) = { data, T_L, T_R }.

where ① T_L is C_{h-1}, T_R is P_{h-2}

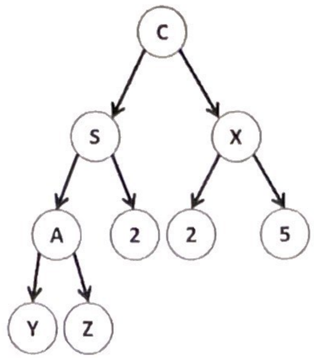
or ② T_L is P_{h-1}, T_R is C_{h-1}

Trees are nothing new – they're fancy linked lists:



(55) (56)

Theorem: If there are n data items in our representation of a binary tree, then there are $(n+1)$ NULL pointers.

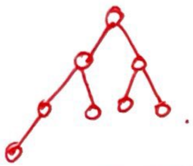


Is every full tree complete?

No.

Is every complete tree full?

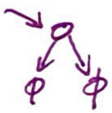
No.



Base Cases:

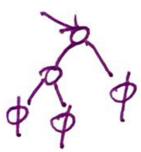
$n=0$: $\rightarrow \phi$ 1 NULL pointer

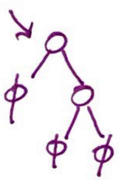
$n=1$:



2 NULL pointers

$n=2$:

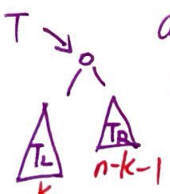




3 NULL pointer

Induction hypothesis: $\forall j<n$, A tree with j data has $(j+1)$ NULL pointers

Consider an arbitrary tree T containing n data elements.

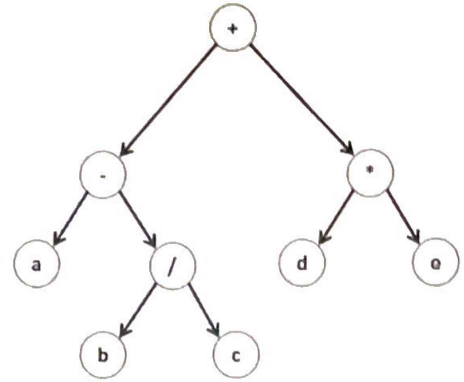


assume T_L has k data, then T_R has $(n-k-1)$ data.

\# of NULL pointers in T = ___ in T_L + ___ in T_R

$= (k+1) + (n-k-1+1) = n+1$.

Traversals:



CS 225 – Things To Be Doing:

1. Programming Exam A is on-going (final day is today!)
2. MP3 extra credit deadline is Monday!
3. lab_quacks due Sunday
4. Daily POTDs

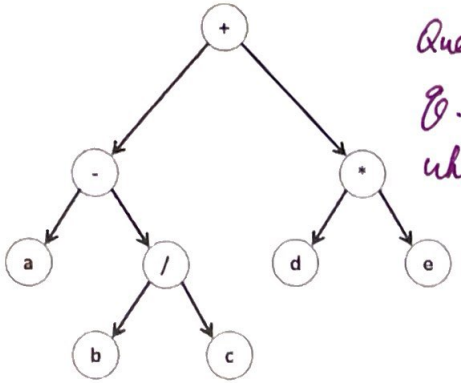
57 58

CS 225

#15: Tree Traversals

February 19, 2018 · *Wade Fagen-Ulmschneider*

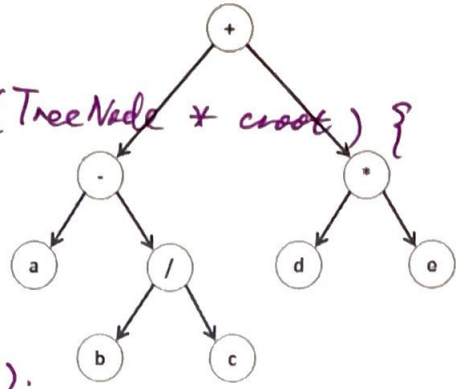
Traversals:



Level Traversal:

1. Create an empty queue.
2. Add the root to the queue.
3. While queue !Empty:
   - 3a. dequeue the queue;
   - 3b. process;
   - 3c. add children to queue;

Thinking Reclusively:




```
template <typename T>.
void BinaryTree<T>::levelOrder(TreeNode * croot) {
Queue <TreeNode * > q;
q.enqueue (croot);
while(!q.empty()) {
TreeNode *t = q.dequeue();
process(t);
q.enqueue(t->left);}
q.enqueue(t->right);}}
```

```
if( t != NULL){
if (t->left != NULL){
if (t->right != NULL){.
```

Or

BinaryTree.cpp

```
TreeNode * BinaryTree<T>::_copy(TreeNode * croot) {
  if (croot != NULL) {
    TreeNode *t = new TreeNode(croot->data);
    t->left = _copy(croot->left);
    t->right = _copy(croot->right);
  }
}
return t;
```

BinaryTree.cpp

```
void BinaryTree<T>::_clear(TreeNode * croot) {
  if (croot != NULL) {
    _clear(croot->left);
    _clear(croot->right);
  } delete croot;
}
croot = NULL;
```

```
else {
return NULL;}
```

59

⑤ + - a / b * d c e. (60)

A Different Type of Traversal

Strategy:

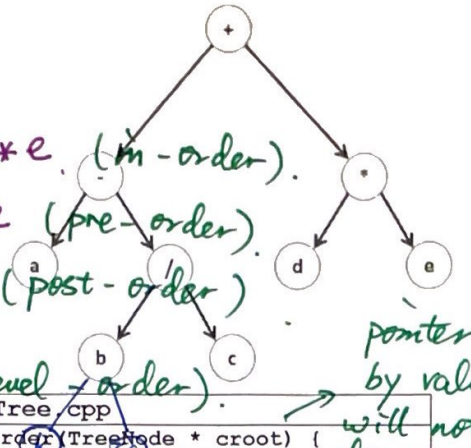
① L curr R : a-b/c+d*e (in-order).

② curr ~~RL~~ LR : + - a / bc * de (pre-order).

③ LR curr : a b c / - d e * + (post-order)

"Reverse Polish"

④ + - * a / d e b c (level-order).



pointer passed by value will not change the original copy.

BinaryTree.cpp

```
void BinaryTree<T>::levelOrder(TreeNode * croot) {
  if (croot != NULL) {
    InOrder(croot->left);
    process(croot);
    InOrder(croot->right);
  }
}
```

InOrder.

```
InOrder(croot->left);
process(croot);
InOrder(croot->right);
```

PostOrder.

```
PostOrder(croot->left);
PostOrder(croot->right);
process(croot);
```

preOrder.

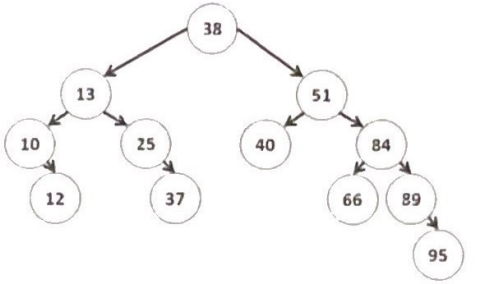
```
process(croot);
preOrder(croot->left);
preOrder(croot->right);
```

Dictionary ADT

Dictionary.h

```
1  #ifndef DICTIONARY_H
2  #define DICTIONARY_H
3
4
5  class Dictionary {
6    public:
7
8
9
10
11
12
13   private:
14
15
16 #endif
```

A Searchable Binary Tree?



BST.h

```
  private:
```

Traversal vs. Search:

Traversal : visit every node exactly once.

Search : find a data in a tree → at least as fast as a traversal

Breadth First Search:

Depth First Search:

CS 225 – Things To Be Doing:

1. Theory Exam 2 Topics List Posted (exam next week)
2. MP3 extra credit deadline tonight
3. Upcoming Lab: lab_trees
4. Daily POTDs

CS 225

#16: Binary Search Tree

February 21, 2018 · *Wade Fagen-Ulmschneider*

Traversal vs. Search:

- **Traversal** visits every node in the tree exactly once.
- **Search** finds one element in the tree.

Breadth First Traversal + Search:

- Look at our children before any other node. → "level order traversal".

Depth First Traversal + Search:

- Look deeper, before looking at our siblings. → "pre-order traversal"

Runtime Analysis on a Binary Tree:

- Find an element: Best case? O(1). Worst case? O(n).
- Insertion of a sorted list of elements?
  Best case? O(1) Worst case?
- Running time bound by?

Dictionary ADT Data is often organized into key/value pairs.

keys are unique.

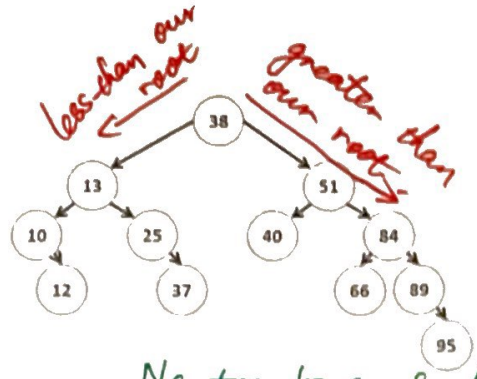
Dictionary.h

```
3
4   template <typename K, typename V>
5   class Dictionary {
6     public:
7
8       V& find(K& key) const;
9       void insert(K& key, V& value);
10      V& remove(K& key);
11
12    private:
13      struct TreeNode {
14
15        TreeNode * left, right;
16        K& key; V& value;
17    }; TreeNode *
```

public function return the value.

private function return the entire tree

A Searchable Binary Tree?



BST.

Binary Search Tree Property: → No two keys of the same value.

$T = \{\}$ or $T = \{d, T_L, T_R\}$ where $x \in T_L$, if $x < d$

$x \in T_R$, if $x > d$.

Finding an element in a BST:

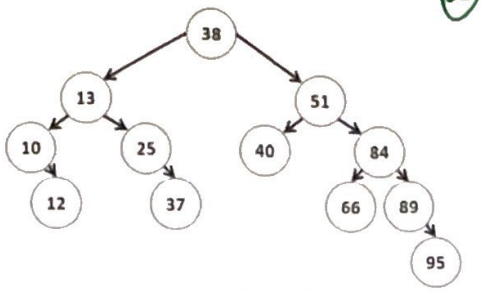
BST.cpp

```
template <typename K, typename V>
TreeNode *& _find(TreeNode *& root, const K & key) const {
  if (key < root->key)
    return _find(root->left, key);
  if (key > root->key)
    return _find(root->right, key);
  if (key == root->key)
    return root;
  if (root == NULL)
    return NULL;
}

V find(const K & key) {
  return _find(root, key)->value;
}
```

what about "NULL" case?

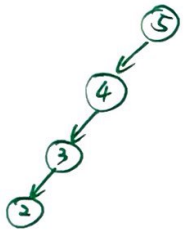
Inserting an element into a BST:



BST.cpp

```
template <class K, class V>
void BST::_insert(TreeNode *& root, K key, V value) {
// ① find the position to insert using _find()
 TreeNode *& t = _find(root, key);
// ② insert node.
 t = new TreeNode(...);
```

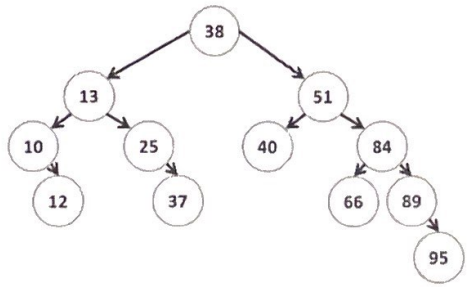
what if



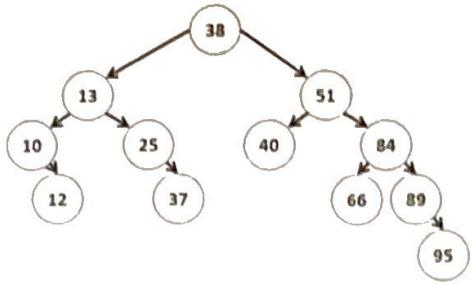

```
}
```

Running time? O(h) **Bound by?** h ≤ n

What if we did not pass a pointer by reference?



Removing an element from a BST:




```
_remove(40)

_remove(25)

_remove(10)

_remove(13)
```

One-child Remove	Two-child remove

BinaryTree.cpp

```
template <class K, class V>
void BST::_remove(TreeNode *& root, const K & key) {




}
```

Running time? _____ **Bound by?** _____

CS 225 – Things To Be Doing:

1. Theory Exam 2 Topics List Posted (exam next week)
2. MP3 extra credit on-going; due Monday, Feb. 26
3. Upcoming Lab: lab_trees
4. Daily POTDs

CS 225

#17: BST Remove

February 23, 2018 · *Wade Fagen-Ulmschneider*

t is a stack variable. not able to update the tree.

BST.cpp

```
template <class K, class V>
void BST::_insert(TreeNode *& root, K & key, V & value) {
  TreeNode *t = _find(root, key);
  t = new TreeNode(key, value);
}
```

insert at leaf.

TreeNode *&t

Running time? _____ **Bound by?** _____

What happens when we run the bugged code above?

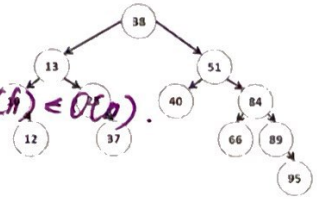
operation — worst case.

find — O(h) ≤ O(n)

insert — _find() + O(1) → O(h) ≤ O(n).

delete — O(h)

traversal



How do we fix the code?

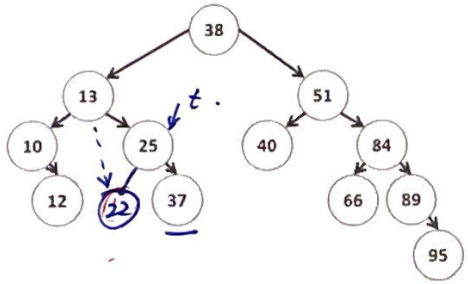
Removing an element from a BST:

_remove(40)

_remove(25)

_remove(10)

_remove(13)



One-child Remove	Two-child remove
In-order predecessor (IoP). – largest node in the left sub-tree. right-most node in left-subtree, with zero or one child.	① _find(13) ② find its IoP. ③ swap n w/ IoP. ④ return n.

BinaryTree.cpp

```
template <class K, class V>
void BST::_remove(TreeNode *& root, const K & key) {
  TreeNode *&t = _find(root, key);
  if (t->left == NULL && t->right == NULL) {
    delete t;
    t = NULL;
  }
  if (t->left != NULL && t->right == NULL) {
    TreeNode *tmp = t;
    t = t->left;
    delete tmp;
  }
}
```

*&t, could change the tree itself.

BST Analysis:
Every operation we have studied on a BST depends on:

...what is this in terms of the amount of data, **n**?

The relationship between the height (h) and size (n):

lower bound: O(lg(n)) upper bound: O(n).

Q: Prove the maximum number of nodes (**n**) given a tree of height **h**.

$2^{h+1}-1$?

$m(h) :=$ maximum # of nodes

$$= \begin{cases} 0, & h = -1 \\ 1 + 2m(h-1), & h > -1. \end{cases}$$

$$= 2^{h+1} - 1.$$

Q: Prove the minimum number of nodes (**n**) in tree of height **h**?

operation	BST Avg. Case	BST Worst Case	Sorted Array	Sorted List
find				
insert				
delete				
traverse				

Final BST Analysis
For every height-based algorithm on a BST:

Lower Bound:

Upper Bound:

Why use this over a linked list?

Q: How does our data determine the height?

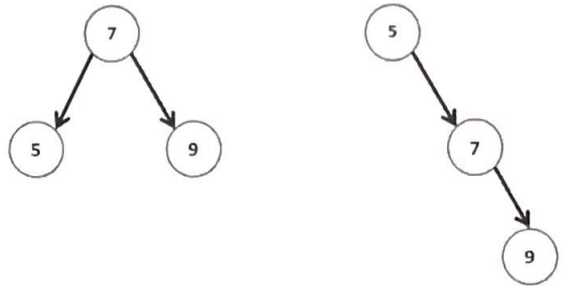
1 3 2 4 5 7 6 vs. 4 2 3 6 7 1 5

Q: How many different ways are there to insert data into a BST?

Q: What is the average height of every arrangement?

Height Balance on BST

What tree makes you happier?



We define the **height balance** (b) of a BST to be:

We define a BST tree T to be **height balanced** if:

CS 225 – Things To Be Doing:

1. Theory Exam 2 starts next Tuesday (topic list is online)
2. MP3 due Monday, Feb. 26; MP4 released on Tuesday
3. lab_trees is due Sunday, Feb. 25
4. Daily POTDs

CS 225

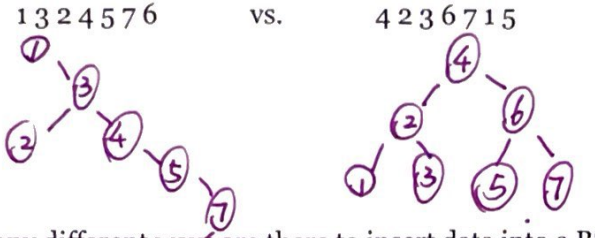
#18: BST Balance

February 26, 2018 · *Wade Fagen-Ulmschneider*

Binary Search Tree (BST) Finale

Q: How does our data determine the height? lower bound $O(\lg(n))$ upper bound $O(n)$.

1 3 2 4 5 7 6 vs. 4 2 3 6 7 1 5



Q: How many different ways are there to insert data into a BST?

$n!$ different orders to insert.

among $n!$ → exactly 2 that creates a linked list.

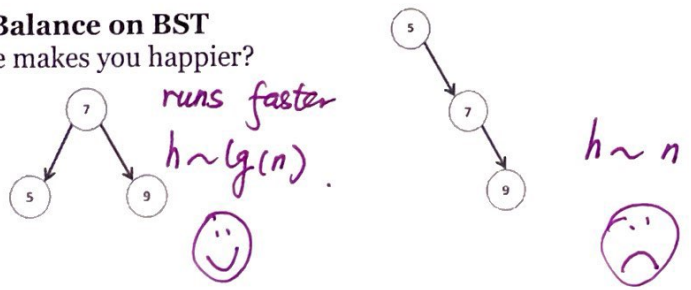
Q: What is the average height of every arrangement?

$\lg(n)$. STAT 420.

operation	BST Avg. Case	BST Worst Case	Sorted Array	Sorted List
find	$O(h) \to O(\lg n)$	$O(h) \to O(n)$	$O(\lg n)$	$O(n)$
insert	- find + insert $O(\lg n) + O(1) = O(\lg n)$	$O(n)$	- find + move $O(\lg n) + O(n) = O(n)$	- find + move $O(n) + O(1) = O(n)$
delete	$O(\lg n)$	$O(n)$	$O(n)$	$O(n)$
traverse	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Height Balance on BST

What tree makes you happier?



Let us describe the **balance (b)** of a BST to be:

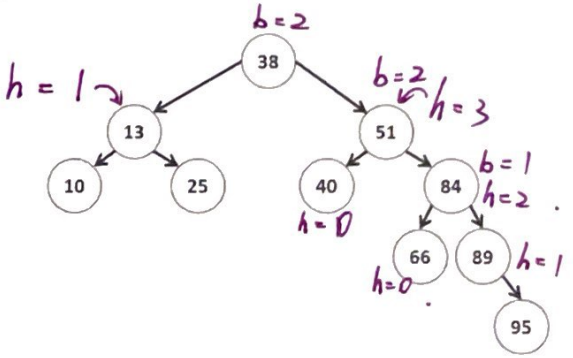
$b = \text{height}(T_R) - \text{height}(T_L)$.

- If **b** is negative: More nodes on left.
- If **b** is positive: More nodes on right.

We define a BST tree T to be **height balanced** if:

$|b| \leq 1$.

A node is considered to be **out of balance** it's not height balanced.
What is the lowest node that is out of balance?



Brining a tree back into balance

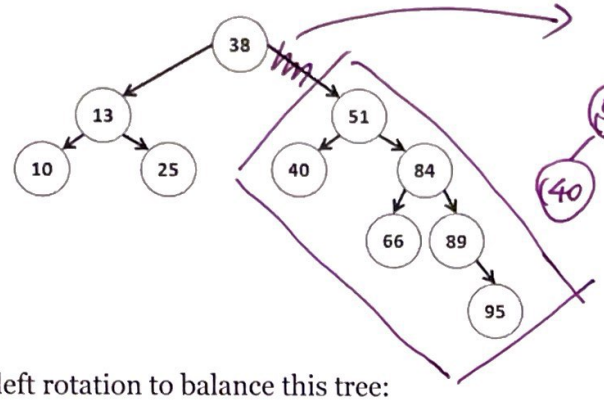
Goal: Create a strategy to bring a BST back into balance after an operation has caused the three to be out of balance.

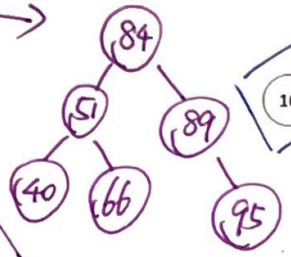
A **Tree Rotation** is an operation that maintains two properties:

1. Turn "Sticks" into "Mountains".
2. Maintain the BST order property.

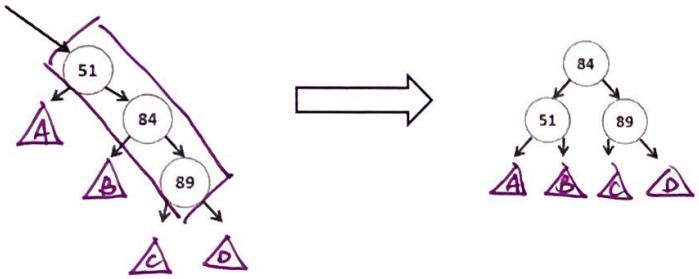
Example: Defining a Rotation

1. Where is the deepest point of imbalance in the following tree?

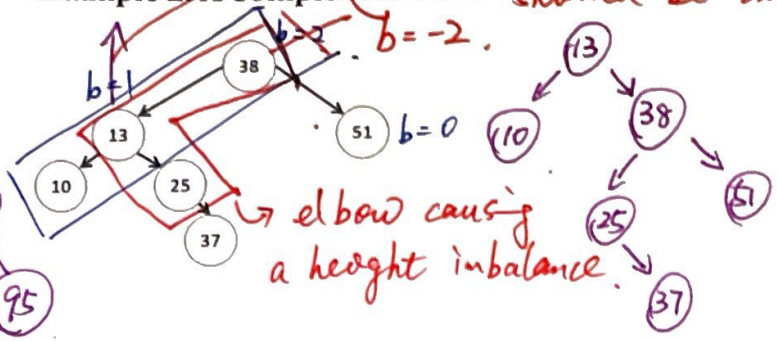




2. Perform a left rotation to balance this tree:

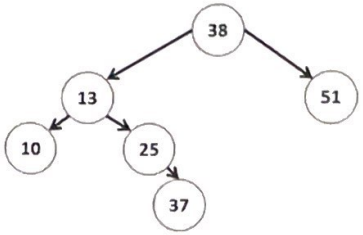


Example 2: A Complex Rotation

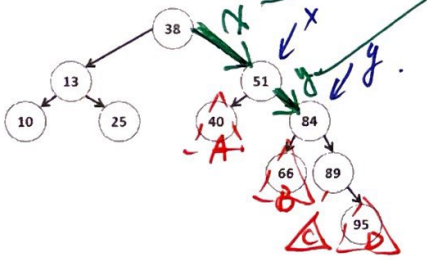


if opposite sign should be an elbow

elbow causing a height imbalance



Implementing a left rotation:



pointers by references.

```
x->right = y->left;
y->left = x;
```

if x & y are references/pointers by.

x = y

running time for 1 rotation is O(1).

BST Rotation Summary:

1. Four kinds of rotations (L, R, LR, and RL)
2. All rotations are local
3. All rotations run in constant time, O(1)
4. BST property is maintained!

Overall Goal:

...and we call these trees:

CS 225 – Things To Be Doing:

1. Theory Exam 2 starts next Tuesday (topic list is online)
2. MP3 due Monday, Feb. 26; MP4 released on Tuesday
3. lab_trees is due Sunday, Feb. 25
4. Daily POTDs

CS 225

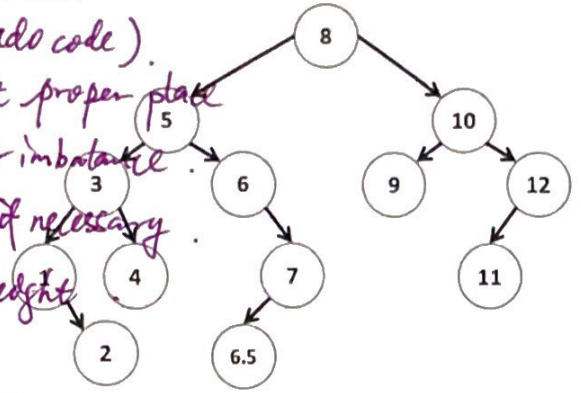
#20: AVL Analysis

March 2, 2018 · *Wade Fagen-Ulmschneider*

AVL Insertion

Insert (pseudo code).

1. insert at proper place
2. check for imbalance
3. rotate, if necessary
4. update height



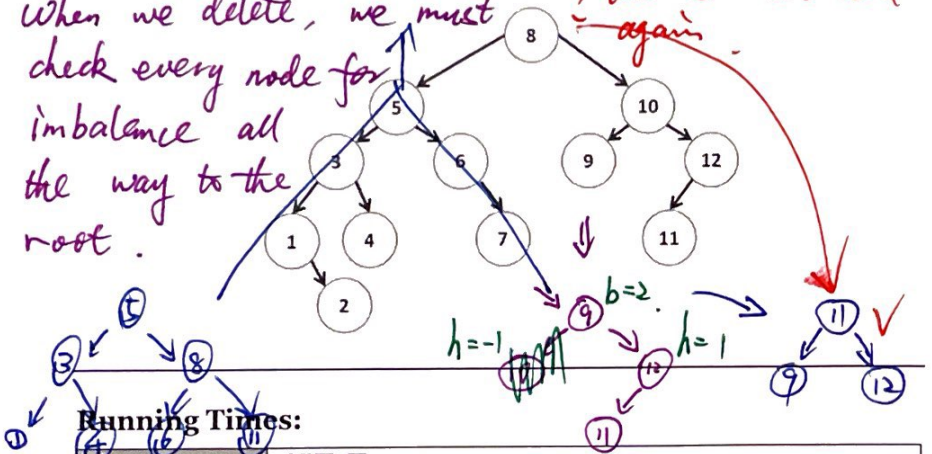
where is the code?

AVL Removal

When we delete, we must check every node for imbalance all the way to the root.

b = -2 → single rotation

→ this is imbalanced again



Running Times:

	AVL Tree
find	O(h) + No Rotations
insert	O(h) + up to 1 rotation
remove	O(h) + up to h rotations.

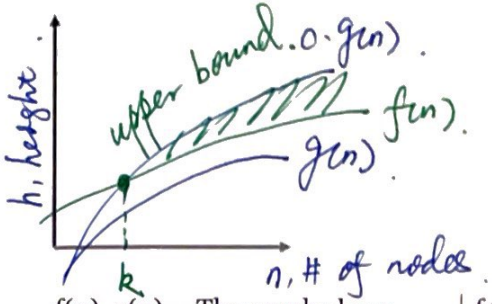
will argue that $h = O(\lg(n))$.

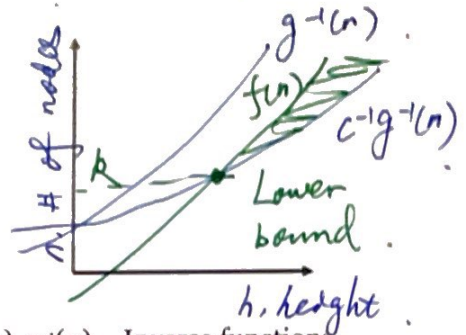
Motivation:

Big-O is defined as: $f(n)$ is $O(g(n))$ iff:

$\exists\, c, k$ s.t. $f(n) \le c \cdot g(n)$ $\forall\, n > k$.

Visually:





f(n), g(n) -- The graph above describes functions of the height (**h**) of an AVL tree given the number of nodes (**n**).

$f^{-1}(n)$, $g^{-1}(n)$ – Inverse functions describe the number of nodes in a tree (**n**) given a height (**h**).

$N(h) = \alpha N(h-1) + \beta$

$= \gamma(N(h-1) - \alpha N(h-2) + \beta)$

Plan of Action:

Goal: Find a function that defines the lower bound on **n** given **h**.

Given the goal, we begin by defining a function that describes the smallest number of nodes in an AVL of height **h**:

We know: insert, find, remove runs O(h) time

we argue that $h = O(\lg n)$

build minimum AVL tree

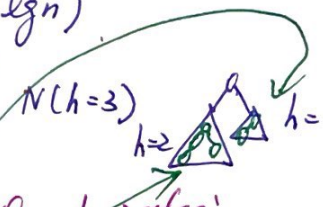
$N(h)$.

$N(h=-1)$: 0 tree

$N(0)$: 1

$N(h=1)$: 2.

$N(h=2)$: 4.



Our hypothesis

$N(h) = 1 + N(h-1) + N(h-2)$

$h > 0$.

7.

State a Theorem:
An AVL tree of height **h** has at least $N(h) = 2^{\frac{h}{2}}$ where $h > 0$.

I. Consider an AVL tree and let **h** denote its height.

II. Case: $h = 1$.
Definition AVL tree:
$\Rightarrow$ 2 nodes $> 2^{\frac{h}{2}} = 2^{\frac{1}{2}} = \sqrt{2} \approx 1.44$

III. Case: $h = 2$
Definition AVL tree:
$\Rightarrow$ 4 nodes $> 2^{\frac{h}{2}} = 2^{\frac{2}{2}} = 2$

IV. Case: _____

By an inductive hypothesis (IH):
$N(h) = N(h-1) + N(h-2)$. $\forall j < n, N(j) > 2^{\frac{j}{2}}$

We show that:

$N(h) > N(h-1) + N(h-2)$
$> 2N(h-2)$
$> 2 \cdot 2^{\frac{h-2}{2}}$
$> 2^{\frac{h}{2}}$

一往无前虎山行
拔开云雾见光明
樊里花开灶母亭
幼象成其歌舞升平

V. Using a proof by induction, we have shown that:
$n \geq N(h) > 2^{\frac{h}{2}} \Rightarrow n > 2^{\frac{h}{2}}$

...and by inverting our finding:

$\lg(n) > \frac{h}{2}$ $\quad h < 2\lg(n)$

$2\lg(n) > h$ $\quad \Rightarrow h = O(\lg(n))$.

Summary of Balanced BSTs:

Advantages	Disadvantages

Iterators + Usefulness
Three weeks ago, you saw that you can use an iterator to loop through data:

```
1  DFS dfs(...);
2  for ( ImageTraversal::Iterator it = dfs.begin();
          it != dfs.end(); ++it ) {
3    std::cout << (*it) << std::endl;
4  }
```

You will use iterators extensively in MP4, creating them in Part 1 and then utilizing them in Part 2. Given the iterator, you can use the for-each syntax available to you in C++:

```
1  DFS dfs(...);
2  for ( const Point & p : dfs ) {
3    std::cout << p << std::endl;
4  }
```

The exact code you might use will have a generic `ImageTraversal`:

```
1  ImageTraversal & traversal = /* ... */;
2  for ( const Point & p : traversal ) {
3    std::cout << p << std::endl;
4  }
```

CS 225 – Things To Be Doing:

1. Theory Exam 2 is ongoing!
2. MP4 extra credit submission ongoing – due Monday, March 5th!
3. lab_huffman is due on Sunday, March 4th
4. Daily POTDs are ongoing!

CS 225

#21: AVL Applications

March 5, 2018 · *Wade Fagen-Ulmschneider*

AVL – Proof of Runtime

On Friday, we proved an upper-bound on the height of an AVL tree is **2*lg(n)** or **O(lg(n))**.

AVL Trees	Red-Black Trees
Balanced BST Max height: 1.44 * lg(n) *Q: Why is our proof 2*lg(n)?* Rotations: - find: 0 rotation - insert: up to 1 rotation - remove: up to h rotation.	Balanced BST *Functionally equivalent to AVL trees; all key operations runs in O(h) time.* Max height: 2 * lg(n) Rotations: - Constant number of rotations for each operation

*In CS 225, we learned **AVL trees** because they're intuitive and I'm certain we could have derived them ourselves given enough time. A red-black tree is simply another form of a balanced BST that is also commonly used.*

Summary of Balanced BSTs:

(Includes both AVL and Red-Black Trees)

Advantages	Disadvantages
- Running time $O(\lg(n))$. improvement over: arrays, linked list, most linear data struc. - Great for specific applications: key not exactly known • nearest neighbor • range	- Running time: Not O(1). Hashing solves finding exact key in near O(1) time. better algorithm. - In-memory requirement. All data must be in main memory not great for big data.

Using a Red-Black Tree in C++

C++ provides us a balanced BST as part of the standard library:

```
std::map<K, V> map;
```

`std::map<std::string, int> Map;` `Map["cs225"] = 4;`

The map implements a dictionary ADT. Primary means of access is through the overloaded `operator[]`:

```
V & std::map<K, V>::operator[]( const K & )
```

This function can be used for both insert and find!

Removing an element:

```
std::map<K, V>::erase( const K & );
```

Range-based searching:

```
iterator std::map<K, V>::lower_bound( const K & );
iterator std::map<K, V>::upper_bound( const K & );
```

one past the largest element in the range.

Iterators

Three weeks ago, you saw that you can use an iterator to loop through data:

```
1  DFS dfs(...);
2  for ( ImageTraversal::Iterator it = dfs.begin();
          it != dfs.end(); ++it ) {
3    std::cout << (*it) << std::endl;
4  }
```

You will use iterators extensively in MP4, creating them in Part 1 and then utilizing them in Part 2. Given the iterator, you can use the for-each syntax available to you in C++:

For ... each loop.

```
1  DFS dfs(...);
2  for ( const Point & p : dfs ) {
3    std::cout << p << std::endl;
4  }
```

The exact code you might use will have a generic `ImageTraversal`:

```
1  ImageTraversal & traversal = /* ... */;
2  for ( const Point & p : traversal ) {
3    std::cout << p << std::endl;
4  }
```

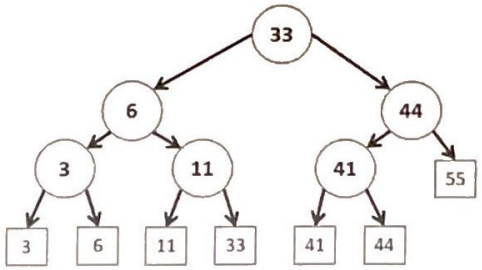
Running Time of Every Data Structure So Far:

	Unsorted Array	Sorted Array	Unsorted List	Sorted List
Find	$O(n)$	$O(\lg(n))$	$O(n)$	$O(n)$
Insert	$O(1)$*	$O(n)$ move ½ elem	$O(1)$	find + $O(1)$ $O(n)$
Remove	$O(n)$	$O(n)$ move ½ elem	find + $O(1)$ $O(n)$	find + $O(1)$ $O(n)$
Traverse	$O(n)$	$O(n)$	$O(n)$	$O(n)$

(data not sorted) (balance not guaranteed)

	Binary Tree	BST	AVL
Find	$O(n)$	$h \leq n$	$h \sim \lg(n)$
Insert	$O(1)$ add at root	$h \leq n$	$h \sim \lg(n)$
Remove	find + $O(1)$ $O(n)$	$h \leq n$	$h \sim \lg(n)$
Traverse	$O(n)$	$O(n)$	$O(n)$.

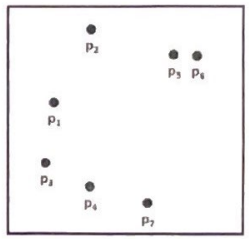
Range-based Searches:



Running Time:

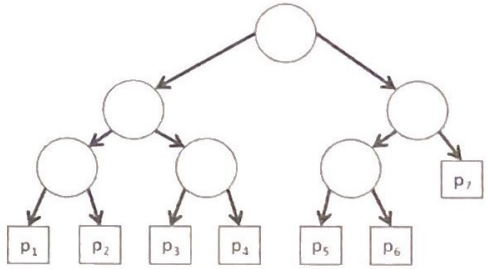
Extending to k-dimensions:

Consider points in 2D: $\mathbf{p} = \{p_1, p_2, \ldots, p_n\}$:



...what points are inside a range (rectangle)?
...what is the nearest point to a query point **q**?

Tree Construction:



Range-based Searches:

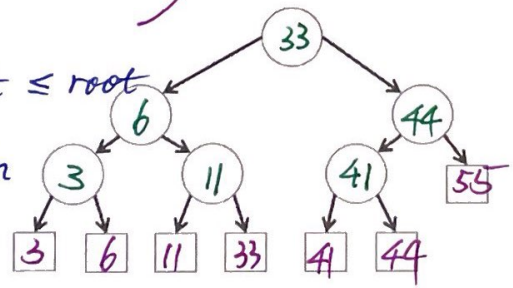
Q: Consider points in 1D: $p = \{p_1, p_2, \ldots, p_n\}$.
...what points fall in [11, 42]?

find upper_bound / lower_bound in $\lg(n)$ time.

Tree Construction:

① T_L ∈ every element ≤ root

② only leaf nodes contain data.



CS 225 – Things To Be Doing:
1. Programming Exam B starts next Tuesday (March 13th)
2. MP4 extra credit due tonight (Monday, March 5th)
3. lab_avl released this week; details regarding labs on Wednesday
4. Daily POTDs are ongoing!

CS 225

#22: kd-Trees and BTrees

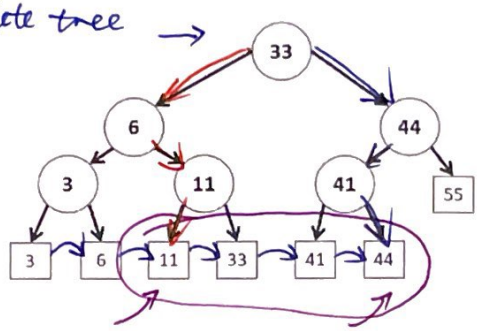
March 7, 2018 · *Wade Fagen-Ulmschneider*

Range-based Searches:

Q: Consider points in 1D: p = {p_1, p_2, ..., p_n}.
...what points fall in [11, 42]?

Tree Construction:

complete tree →



Range-based Searches:

q_low q_high.

[11, 42]
↙ ↘
q_low q_high.

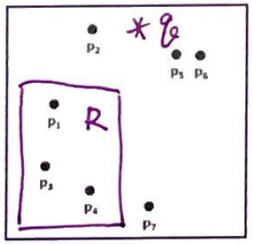
Running Time:

O(lg(n) + k).
k: # of elements in the result set.

~~direct sorting works best for 1D array, not necessarily~~ for high-dimensions.

Extending to k-dimensions:

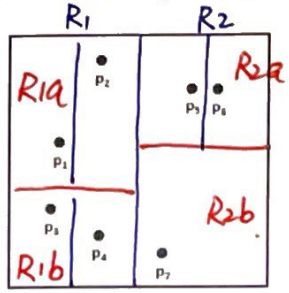
Consider points in 2D: **p = {p_1, p_2, ..., p_n}:**



...what points are inside a range (rectangle)?
...what is the nearest point to a query point **q**?

kd-Tree Motivation:

First, let's try and divide our space up:



① divide |

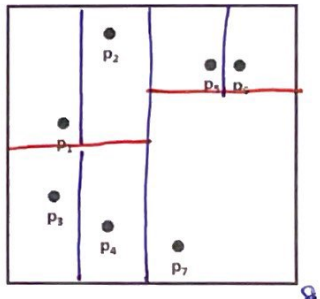
② divide —.

make sure this tree works in every space.

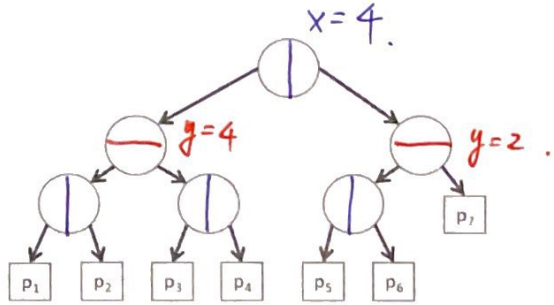
kd-Tree Construction:

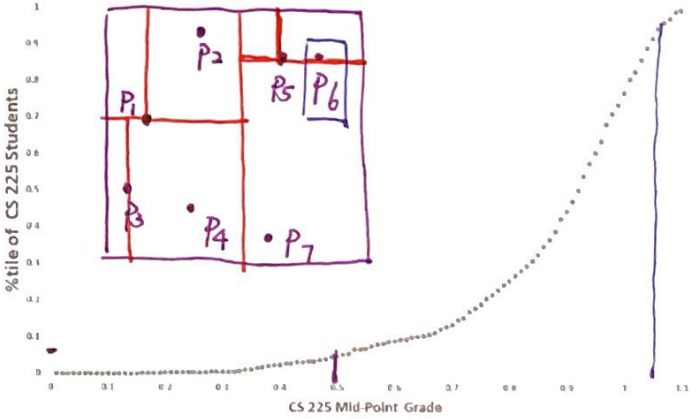
How many dimensions exist in our input space?

How do we want to "order" our dimensions?



8x8.





Motivation

Can we always fit our data in main memory?

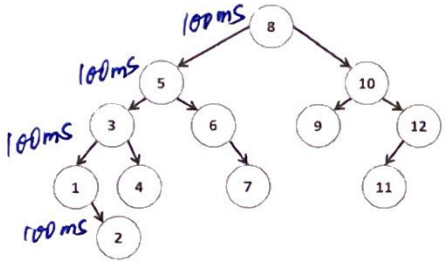
No. ‼

Where else do we keep our data?

- On disk 30–100 ms / load ‼.
- On cloud.

vs. CPU: 3 GHz == 3m ops / 1ms * ___ cores

AVL Operations on Disk:



find 2.

400 ms ‼.

How deep do AVL trees get?

Imagine storing facebook profile.

How many records? 500M

How much data in total? 5 MB / profile picture.

How deep is the AVL tree? h = 29-30 levels.

How much storage? ⇒ 3 sec / look.

2.5 PB.

BTree Motivations

Knowing that we have long seek times for data, we want to build a data structure with two (related) properties:

1. minimal tree.
2. all data we look is relevant.

BTree$_m$

-3	8	23	25	31	42	43	55

m=9

Goal: Build a tree that uses _____ /node!

...optimize the algorithm for your platform!

A **BTree of order m** is an m-way tree where:

1. All keys within a node are ordered.
2. All leaves contain no more than **m-1** nodes.

BTree Insert, using m=5

...when a BTree node reaches **m** keys:

CS 225 – Things To Be Doing:
1. Programming Exam B starts next Tuesday (March 13th) 2. MP4 extra credit ongoing (final deadline March 12th) 3. lab_avl released this week; lab sections **are** being held this week! 4. Daily POTDs are ongoing!

CS 225

#23: BTrees

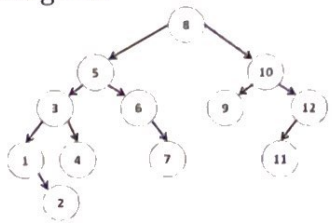
March 12, 2018 · *Wade Fagen-Ulmschneider*

BTree Motivation

Big-O assumes uniform time for all operations, but this isn't always true.

However, seeking data from the cloud may take 100ms+.
...an O(lg(n)) AVL tree no longer looks great:

goes to deep quickly.



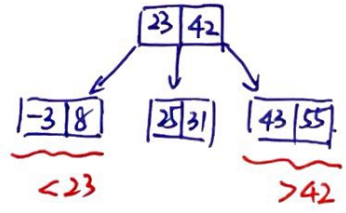
Consider Facebook profile data:

How many profiles?	500 M.	
How much data /profile?	X5MB	
	AVL Tree	BTree
Tree Height	~ 30 levels.	

BTree Motivations

Knowing that we have long seek times for data, we want to build a data structure with two (related) properties:

1. keep the tree short.
2. keep the data relevant.



BTree$_m$

sorted array.

m-1.

m = # of keys that fit in a sorted array.

Goal: Build a tree that uses 1500B 4kB /node!
...optimize the algorithm for your platform!

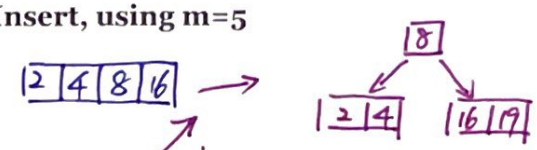
A **BTree of order m** is an m-way tree where:

1. All keys within a node are ordered.

2. all leaves contain hold no more than (m-1) nodes.

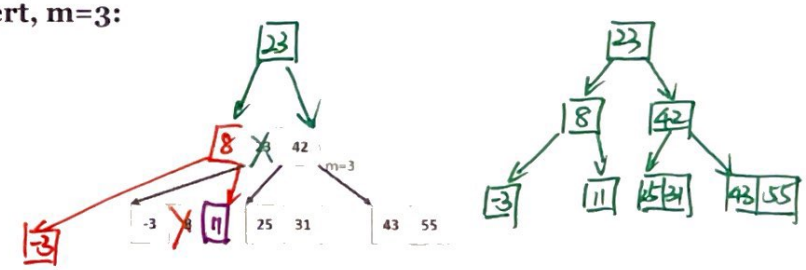
BTree Insert, using m=5

2
4
8
16
19.



...when a BTree node reaches **m** keys:

BTree Insert, m=3:



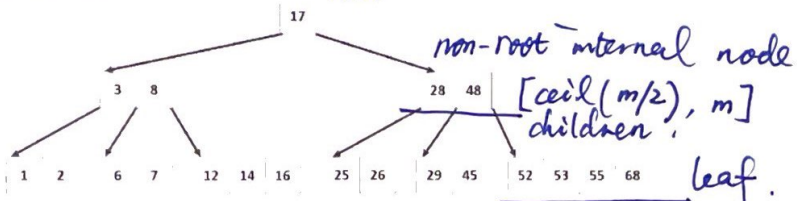
Great interactive visualization of BTrees:
https://www.cs.usfca.edu/~galles/visualization/BTree.html

BTree Properties

For a BTree of order **m**:

1. All keys within a node are ordered.
2. All leaves contain no more than **m-1** ~~nodes~~ keys.
3. All internal nodes have exactly **one more ~~key~~ than ~~children~~**. key child
4. Root nodes can be a leaf or have **[2, m]** children.
5. All non-root, internal nodes have **[ceil(m/2), m]** children.
6. All leaves are on the same level.

Example BTree



What properties do we know about this BTree?

leaf: $4 \le m-1$

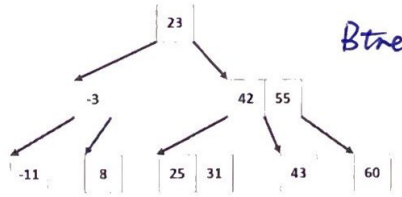
non-root internal: $3 \ge \text{ceil}(m/2)$ $\Rightarrow$ $5 \le m \le 6$

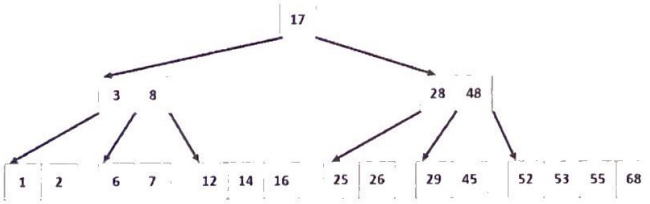
$m = 5$, or 6.

BTree Search

AVL: $\log_2(n)$

Btree: $h \sim \log_m(n)$





BTree.cpp (partial)

```
1  bool Btree::_exists(BTreeNode & node, const K & key) {
2
3    unsigned i;
4    for (i=0; i<node.keys_ct_ && key<node.keys_[i]; i++) {
5  }
6
7    if ( i < node.keys_ct_ && key == node.keys_[i] ) {
8      return true;
9    }
10
11   if ( node.isLeaf() ) {
12     return false;
13   } else {
14     BTreeNode nextChild = node._fetchChild(i);
15     return _exists(nextChild, key);
16   }
   }
```

1 | 10 | 15 | 17 key = 12.

linear search

if we find data.

BTree Analysis

The height of the BTree determines maximum number of _____ possible in search data.

...and the height of our structure:

Therefore, the number of seeks is no more than: _____.

...suppose we want to prove this!

BTree Analysis

In our AVL Analysis, we saw finding an upper bound on the height (given **n**) is the same as finding a lower bound on the nodes (given **h**).

Goal: We want to find a relationship for BTrees between the number of keys (**n**) and the height (**h**).

CS 225 – Things To Be Doing:

1. Programming Exam B starts next tomorrow (March 13th)
2. MP4 due tonight
3. lab_btree released this week; due Tuesday, March 27th at 11:59pm
   (That's the Tuesday evening after spring break)
4. Daily POTDs are ongoing!

CS 225

#24: BTree Analysis + Hashing Intro

March 14, 2018 · *Wade Fagen-Ulmschneider*

BTree Properties

For a BTree of order **m**:

1. All keys within a node are ordered.
2. All leaves contain no more than **m-1** ~~nodes~~ keys.
3. All internal nodes have exactly **one more child than keys**.
4. Root nodes can be a leaf or have **[2, m]** children.
5. All non-root, internal nodes have **[ceil(m/2), m]** children.
6. All leaves are on the same level. [ceil($\frac{m}{2}$) - 1, m - 1] nodes.

BTree Analysis

The height of the BTree determines maximum number of lookups (slow) possible in search data.

...and the height of our structure: $h \sim \log_m n$

m is usually ≥ 100, sometimes ≥ 1000.

Therefore, the number of seeks is no more than: $\log_m(n)$.

...suppose we want to prove this!

BTree Proof #1

In our AVL Analysis, we saw finding an **upper bound** on the height (**h** given **n**, aka **h = f(n)**) is the same as finding a **lower bound** on the keys (**n** given **h**, aka **f^{-1}(n) == g(h)**).

Goal: We want to find a relationship for BTrees between the number of keys (**n**) and the height (**h**).

BTree Strategy:

1. Define a function that counts the minimum number of nodes in a BTree of a given order.
   a. Account for the minimum number of keys per node.
2. Proving a minimum number of nodes provides us with an upper-bound for the maximum possible height.

Proof:

1a. The minimum number of nodes for a BTree of order **m** at each level is as follows:

let $t = \lceil m/2 \rceil$

root: 1 node

level 1: 2 nodes. → minimum t children.

level 2: $2t$ nodes (t-1) keys for each node

level 3: $2t^2$ nodes

...

level h: $2t^{h-1}$ nodes

1b. The minimum total number of nodes is the sum of all levels:

$$1 + 2 + 2t + 2t^2 + \cdots + 2t^{h-1}$$

$$1 + \sum_{i=0}^{h-1} 2t^i = 1 + 2\left(\frac{t^h - 1}{t - 1}\right)$$

---

2. The minimum number of keys:

$$1 + 2\left(\frac{t^h - 1}{t - 1}\right)(t - 1) = 2t^h - 1.$$

$$n \geq 2t^h - 1. \quad \frac{n+1}{2} \geq t^h$$

---

3. Finally, we show an upper-bound on height:

$$\log_t\left(\frac{n+1}{2}\right) \geq h$$

$$h \leq \log_t\left(\frac{n+1}{2}\right) = \log_{\lceil m/2 \rceil}\left(\frac{n+1}{2}\right) \sim \log_m(n)$$

m = 101. h = 4.

So, how good are BTrees?
Given a BTree of order 101, how much can we store in a tree of height=4?

Minimum:

$\approx 2^{h}-1 \rightarrow 2(\lceil 101/2 \rceil)^{4} - 1$

$\approx 2.504 - 1$

≈ 12.5 M.

Maximum:

Hashing

AVL look up: log(n).

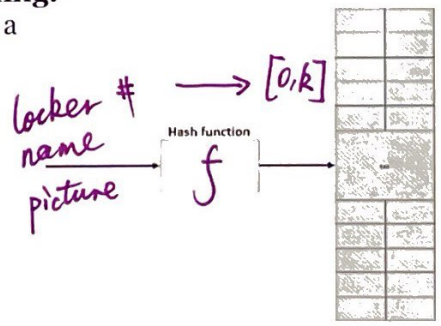
Locker Number	Name
103	Wade
92	Blake
330	
46	
124	

key space.

...how might we create this today?

Goals for Understanding Hashing:

1. We will define a **keyspace**, a (mathematical) description of the keys for a set of data.
2. We will define a function used to map the **keyspace** into a small set of integers.



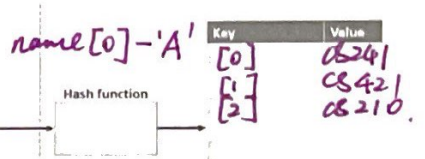
All hash tables consists of three things:

1. A Hash function f
2. An array
3. (??) mystery

A Perfect Hash Function

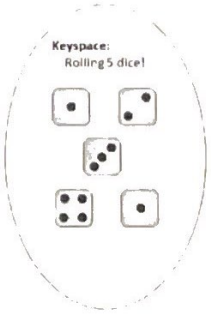
A→0.

(Angrave, CS 241)
(Beckman, CS 421)
(Cunningham, CS 210)
(Davis, CS 101)
(Evans, CS 126)
(Fagen-Ulmschneider, CS 225)
(Gunter, CS 422)
(Herman, CS 233)



...characteristics of this function?

A Second Hash Function



0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

...characteristics of this function?

CS 225 – Things To Be Doing:
1. Programming Exam B is ongoing
2. MP5 has been released; EC deadline is Monday back from break
3. lab_btree released this week; due Tuesday, March 27th at 11:59pm *(That's the Tuesday evening after spring break)*
4. Daily POTDs are ongoing!

CS 225

#25: Hashing

March 16, 2018 · *Wade Fagen-Ulmschneider*

瓦德 费根-奥姆施耐德

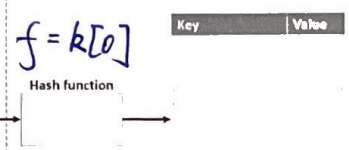
Every hash table contains three pieces:

1. A **hash function, f(k)**. The hash function transforms a key from the keyspace into a small integer.
2. **An array**.
3. **A mystery** third element.

A Perfect Hash Function

Bijective function.

(Angrave, CS 241)
(Beckman, CS 421)
(Cunningham, CS 210)
(Davis, CS 101)
(Evans, CS 126)
(Fagen-Ulmschneider, CS 225)
(Gunter, CS 422)
(Herman, CS 233)

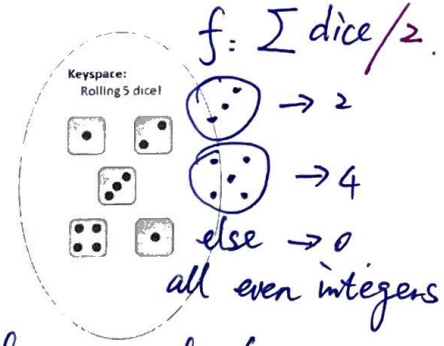


...characteristics of this function?

O(1) time.

A Second Hash Function

better, both odd and even integers.



f: Σ dice/2.

→ 2

→ 4

else → 0

all even integers

petals around the rose

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

...characteristics of this function?

人生就像一张茶几
上面摆满了光剧

All hash functions will consist of two parts:

- A **hash**: f(k) or h(k).
- A **compression**: usually (mod, % array size)

(a) don't create your own (yet*)

(b) very smart people have created very bad hash functions.

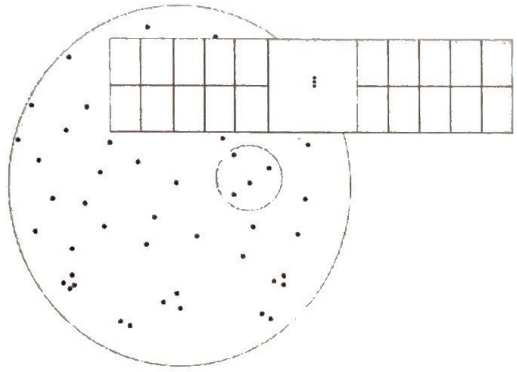
Characteristics of a good hash function:

1. Computation Time:
   must be constant time O(1).
2. Deterministic:
   if ($k_1 == k_2$) $h(k_1) = h(k_2)$
3. SUHA:
   simple uniform hashing assumption.
   $\forall i, j$, $i != j$ $P(h(i) == h(j)) = \frac{1}{N}$.

Towards a general-purpose hashing function:
It is <u>easy to create</u> a general-purpose hashing function when the keyspace is proportional to the table size:

- **Ex:** Professors at CS@Illinois
- **Ex:** Anything you can reason about every possible value

It is difficult to create a general-purpose hashing function when the keyspace is large:



My 40-character strategy:

	Alice in Wonderland, Page 1
1	Alice was beginning to get very tired of
2	sitting by her sister on the bank, and
3	of having nothing to do: once or twice s
4	he had peeped into the book her sister w
5	as reading, but it had no pictures or co
6	nversations in it, 'and what is the use
7	of a book,' thought Alice 'without pictu
8	res or conversations?' So she was consi
9	dering in her own mind (as well as she c
10	ould, for the hot day made her feel very
11	sleepy and stupid), whether the pleasur
12	e of making a daisy-chain would be worth
13	the trouble of getting up and picking t
14	he daisies, when suddenly a White Rabbit
15	with pink eyes ran close by her. There
16	was nothing so very remarkable in that;
17	nor did Alice think it so very much out
18	of the way to hear the Rabbit say to it
19	self, 'Oh dear! Oh dear! I shall be late
20	!' (when she thought it over afterwards,
21	it occurred to her that she ought to ha

"Alice wa" → A

...what is a naïve hashing strategy for this input?

$f = hash(s[0] + s[1] + \cdots + s[7])$ $\alpha = 1$.

...characteristics of this function?

What is an example of bad input data on this hash function?

https://en.wikipedia/wiki/mainpage.

https://en. " " /Vector.

how random is this? all map to one element!

Reflections on Hashing

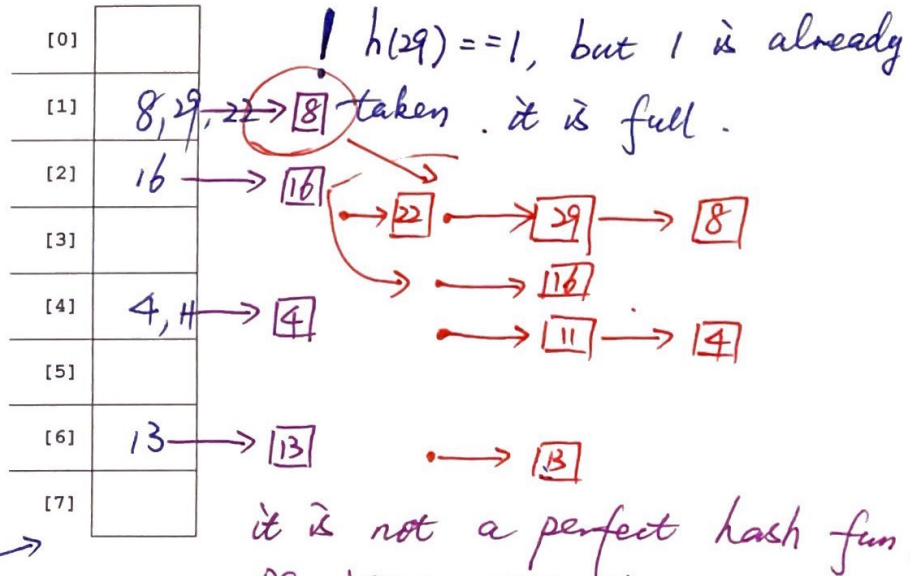
We are starting the study of general-purpose hash functions. There are many other types of hashes for specific uses (ex: cryptographic hash functions).

Even if we build a good hash function, it is not perfect. What happens when the function isn't always a bijection?

chain the elements into a linked list

Collision Handling Strategy #1: Separate Chaining

Example: **S = { 16, 8, 4, 13, 29, 11, 22 }, |S| = n**
h(k) = k % 7, |Array| = N



Load Factor:

$\alpha = \frac{n}{N}$.

Running time of Separate Chaining:

	Worst Case	SUHA
Insert	O(1)	O(1).
Remove/Find	O(n)	O(n/N) → O(α).

CS 225 – Things To Be Doing:
1. Programming Exam B is ongoing
2. MP5 has been released; EC+7 deadline is Monday back from break
3. lab_btree released this week; due Tuesday, March 27th at 11:59pm *(That's the Tuesday evening after spring break)*
4. Daily POTDs are ongoing!

CS 225

#26: Hashing

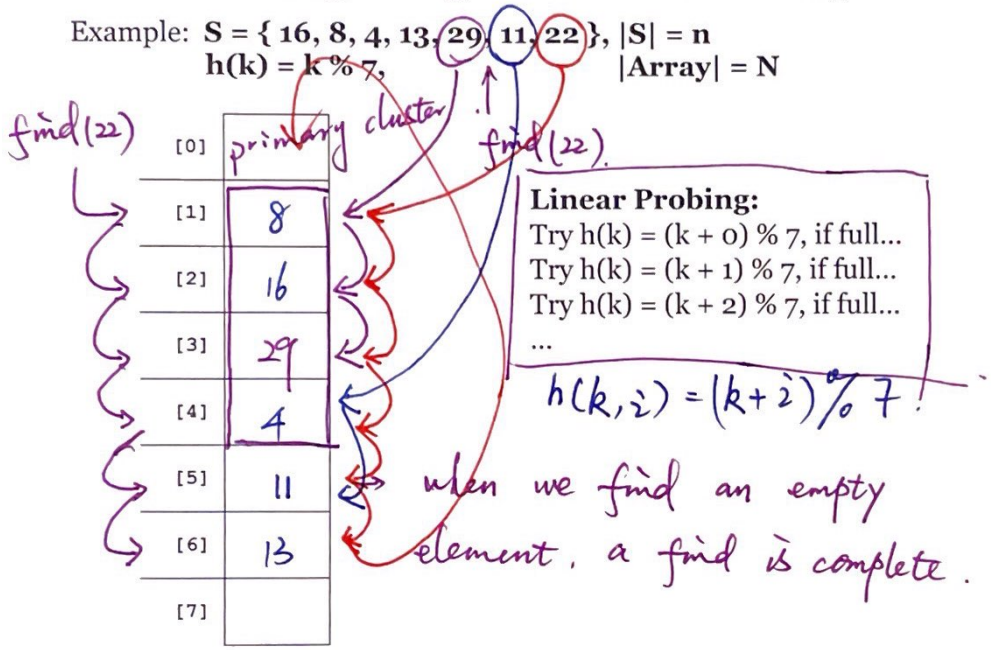
March 26, 2018 · *Wade Fagen-Ulmschneider*

Every hash table contains three pieces:

1. A **hash function**, **f(k)**. The hash function transforms a key from the keyspace into a small integer.
2. **An array**.
3. **A mystery** third element.

Collision Handling Strategy #2: Probe-based Hashing

Example: **S = { 16, 8, 4, 13, 29, 11, 22 }, |S| = n**
h(k) = k % 7, |Array| = N



Linear Probing leads to Primary Clustering

Description: large sequential block of data, resulting in long search times.

Remedy: increase each step in a non-linear way.

Double Hashing:

Example: **S = { 16, 8, 4, 13, 29, 11, 22 }, |S| = n**
$h_1(k)$ = k % 7, $h_2(k)$ = 5 − (k % 5), |Array| = N

[0]	
[1]	8
[2]	16
[3]	29
[4]	4
[5]	11
[6]	13
[7]	

29 % 7 = 1, collision. $h_2(k) = 5 - 4 = 1$.

Double Hashing:
Try h(k) = (k + 0*h_2(k)) % 7, if full...
Try h(k) = (k + 1*h_2(k)) % 7, if full... step.
Try h(k) = (k + 2*h_2(k)) % 7, if full... (29+1) % 7
...
(29+1x2) % 7.

h(k, i) = (h_1(k) + i*h_2(k)) % 7

Running Time:

Linear Probing:
- Successful: $\frac{1}{2}(1 + 1/(1-\alpha))$
- Unsuccessful: $\frac{1}{2}(1 + 1/(1-\alpha))^2$

Double Hashing:
- Successful: $1/\alpha * \ln(1/(1-\alpha))$
- Unsuccessful: $1/(1-\alpha)$

Separate Chaining:
- Successful: $1 + \alpha/2$
- Unsuccessful: $1 + \alpha$

α: load factor. $\frac{n}{N}$

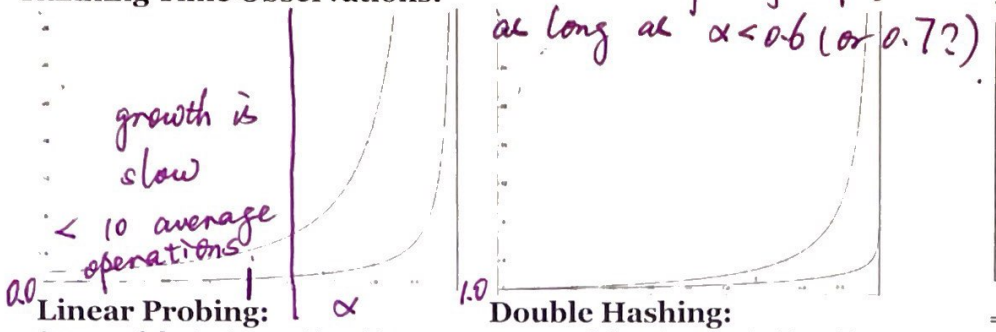
$\alpha = \frac{\text{\# of elements}}{\text{array size}}$

Running Time Observations:

1. As α increases: our running time increases.
2. If α is held constant: running time stays the same.

Running Time Observations:

O(1) time for find as long as $\alpha < 0.6$ (or 0.7?)



Linear Probing:
Successful: $\frac{1}{2}(1 + 1/(1-\alpha))$
Unsuccessful: $\frac{1}{2}(1 + 1/(1-\alpha))^2$

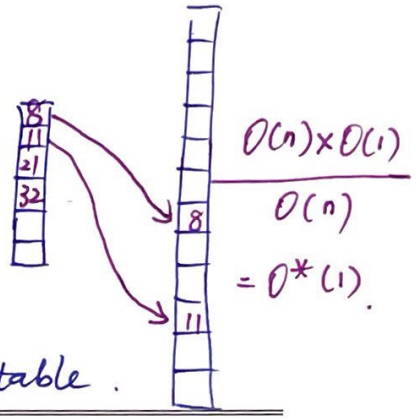
Double Hashing:
Successful: $1/\alpha * \ln(1/(1-\alpha))$
Unsuccessful: $1/(1-\alpha)$

Analysis of Dictionary-based Data Structures

	Hash Table		AVL	List
	Amortized	Worst Case		
Find				
Insert				
Storage Space				

A Secret, Mystery Data Structure:

ReHashing:
What happens when the array fills?
① double size ② copy data.
- ensure a table of appropriate size
Better question: what if $\alpha \geq 0.6$



$O(n) \times O(1)$
$O(n)$
$= O*(1)$.

Algorithm:
2A: update the hash function
2B: rehash all values in our table.

ADT:
insert
remove
isEmpty

Which collision resolution strategy is better?
- Big Records: separate chaining (open hashing).
- Structure Speed: double hashing (closed hashing).

What structure do hash tables replace?
Dictionary

What constraint exists on hashing that doesn't exist with BSTs? only find exact key matches.

Why talk about BSTs at all?
range finding is important.

CS 225 – Things To Be Doing:

1. Theory Exam 3 starts next week (Tuesday, April 3rd)
2. MP5 has been released; EC+7 deadline is tonighit
3. lab_btree due this Tuesday, March 27th at 11:59pm
4. Daily POTDs are ongoing!

CS 225

#27: Heaps

March 28, 2018 · *Wade Fagen-Ulmschneider*

simple uniform hashing assumption.

chaining: all hashed to the same slot, a long linked list.

Analysis of Dictionary-based Data Structures

	Hash Table		AVL	List
	SUHA	Worst Case		
Find	O(1)* with a constant load factor.	O(n)	O(h)=O(log n)	O(n).
Insert	O(1)* $\alpha < 0.7$	O(n)	O(h)=O(log n)	O(1).
Storage Space	n/α. O(n)		n(data + 2x ptr) O(n)	n(data + ptr) O(n).

Implementation of priority queue.

insert	removeMin	Implementation
~~O(n)~~ O(1).	O(n) ✓	Unsorted Array
O(1)	O(n) ✓	Unsorted List
~~O(lg(n))~~ O(n) ✓	O(1)	Sorted Array
~~O(lg(n))~~ O(n). ✓	O(1)	Sorted List

Q1: What errors exist in this table? (Fix them!)

Q2: Which algorithm would we use?

Data Structures in std library:

- std::map → balanced BST. O(log n).
- std::unordered_map → Hash table.

:: load_factor().

:: max_load_factor(ml). → how about change ml to 1.0.

A Secret, Mystery Data Structure:

could store

could not store:

CS225: sphere.
hues
saturation
alpha.

Ordered.

Heap:

- dogs.
- cats.
- BST.

No clear order.

ADT:

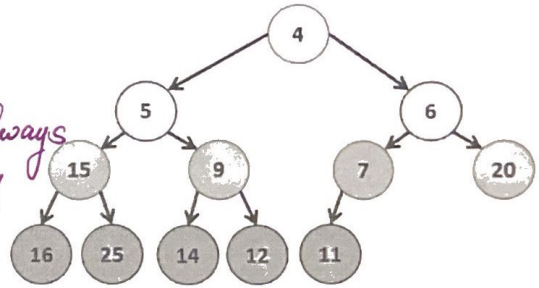
insert

remove min/max.

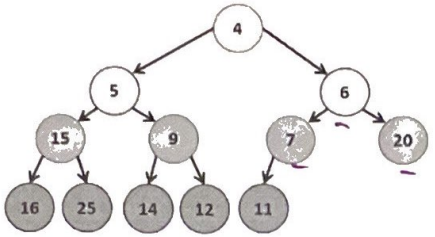
isEmpty

A New Tree-like Structure:

- binary tree
- parents are always smaller than all descents.
- complete tree.
- recursive structure.



Implementing a (min)Heap as an Array



X	4	5	6	15	9	7	20	16	25	14	12	11			

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8 9

leftChild(index): ~~2i+1~~ 2i

rightChild(index): ~~2i+2~~ 2i+1

parent(index): $\lfloor \frac{i-1}{2} \rfloor$ $\lfloor \frac{i}{2} \rfloor$

A complete binary tree T is a min-heap if:

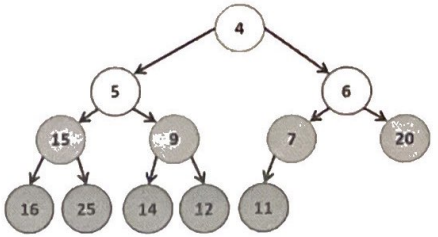
-
-

_elements

X					

0 1 2 3 4 5 5

Insert:



-	4	5	6	15	9	7	20	16	25	14	12	11			

Heap.cpp (partial)

```
1  template <class T>
2  void Heap<T>::_insert(const T & key) {
3    // Check to ensure there's space to insert an element
4    // ...if not, grow the array
5    if ( size_ == capacity_ ) { _growArray(); }
6
7    // Insert the new element at the end of the array
8    item_[++size] = key;
9
10   // Restore the heap property
11   _heapifyUp(size);
12 }
```

```
31 template <class T>
32 void Heap<T>::_heapifyUp( _____ ) {
33   if ( index > _____ ) {
34     if ( item_[index] < item_[ parent(index) ] ) {
35       std::swap( item_[index], item_[ parent(index) ]
36 );
37       _heapifyUp( _____ );
38     }
39   }
40 }
```

CS 225 – Things To Be Doing:

1. Theory Exam 3 starts next week (Tuesday, April 3rd)
2. MP5 deadline is Monday, April 2nd.
3. lab_hash released today; due Sunday, April 1st
4. Daily POTDs are ongoing!

CS 225

#28: Heap Operations

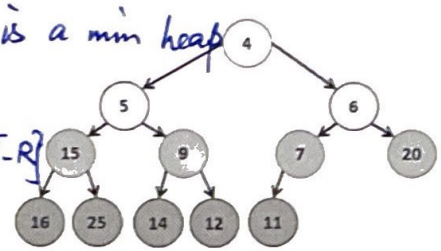
March 30, 2018 · *Wade Fagen-Ulmschneider*

A Heap Data Structure

(specifically a minHeap in this exampl, as the minimum element is at the root)

complete BST is a min heap if $T = \{\}$

$T = \{r, T_L, T_R\}$ where T_L, T_R are both min heaps and r is smaller than the root of T_L and T_R.



-	4	5	6	15	9	7	20	16	25	14	12	11			

Given an index i, it's parent and children can be reached in O(1) time:

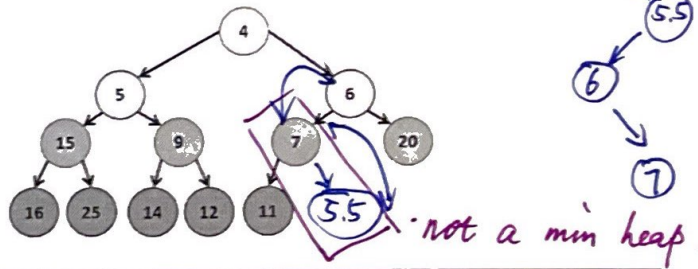
- leftChild := 2i
- rightChild := 2i + 1
- parent := floor(i / 2)

Formally, a complete binary tree T is a minHeap if:

- $T = \{\}$
- $T = \{r, T_L, T_R\}$ where T_L, T_R are both min heaps and r is smaller than the root of T_L and T_R.

Inserting into a Heap

insert (5.5)



not a min heap.

-	4	5	6	15	9	7	20	16	25	14	12	11			

Heap.cpp (partial)

```
1  template <class T>
2  void Heap<T>::_insert(const T & key) {
3    // Check to ensure there's space to insert an element
4    // ...if not, grow the array
5    if ( size_ == capacity_ ) { _growArray(); }
6
7    // Insert the new element at the end of the array
8    item_[++size] = key;
9
10   // Restore the heap property
11   _heapifyUp(size);
12 }
31 template <class T>
32 void Heap<T>::_heapifyUp( int index ) {
33   if ( index > 1 ) {
34     if ( item_[index] < item_[ parent(index) ] ) {
35       std::swap( item_[index], item_[ parent(index) ]
36 );
37       _heapifyUp( parent(index) );
38     }
39   }
40 }
```

O(1)*

double the size. copy the data.

O(1).

O(log n).

unsigned.

concern of location.

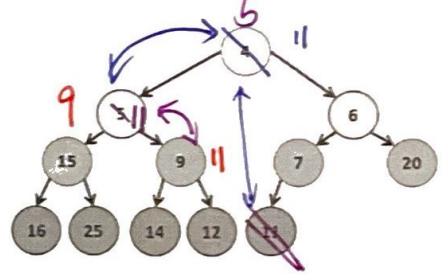
parent(index) = floor(index/2)

What's wrong with this code?

Running time of insert?

_heapifyup(size) → O(h) = O(log n)

Heap Operation: removeMin / heapifyDown:



-	4	5	6	15	9	7	20	16	25	14	12	11			

```
Heap.cpp (partial)
1  template <class T>
2  void Heap<T>::_removeMin() {
3    // Swap with the last value
4    T minValue = item_[1];
5    item_[1] = item_[size_];
6    size--;
7
8    // Restore the heap property
9    heapifyDown();
10
11   // Return the minimum value
12   return minValue;
13 }
```

O(1)*

O(h) check the element in the last half of capacity. = O(lgn)

```
1  template <class T>
2  void Heap<T>::_heapifyDown(int index) {
3    if ( !_isLeaf(index) ) {
4      T minChildIndex = _minChild(index);
5      if ( item_[index] > item_[minChildIndex] ) {
6        std::swap( item_[index], item_[minChildIndex] );
7        _heapifyDown( minChildIndex );
8      }
9    }
10 }
```

← base case

Theorem: The running time of buildHeap on array of size n is: _____.

Strategy:

Define S(h):
Let **S(h)** denote the sum of the heights of all nodes in a complete tree of height **h**.

S(0) =

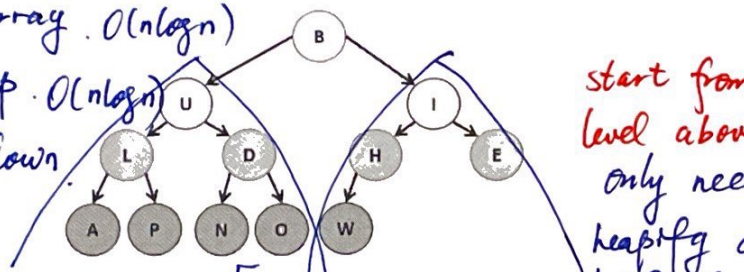
S(1) =

S(h) =

Proof of S(h) by Induction:

Q: How do we construct a heap given data?

1. sorted array. O(nlogn)
2. heapify up. O(nlogn)
3. heapify down.



start from one level above the leaf. only need to heapify down from half of the array

-	B	U	I	L	D	H	E	A	P	N	O	W			

```
Heap.cpp (partial)
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = parent(size); i > 0; i--) {
4      heapifyDown(i);
5    }
6  }
```

assumption: left & right subtree are both heaps.

Running Time?

one-element structure is heap. bottom half of the array are already heap.

Finally, finding the running time:

CS 225 – Things To Be Doing:

1. Theory Exam 3 starts next week (Tuesday, April 3rd)
2. MP5 deadline is Monday, April 2nd
3. lab_hash is due Sunday, April 1st
4. Daily POTDs are ongoing!

Tonight 7pm, 1404SC

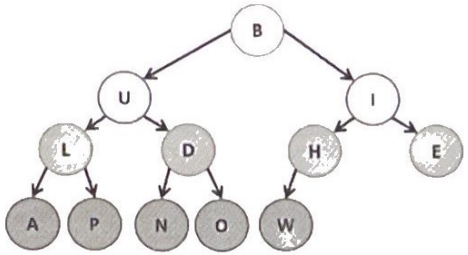
CS 225

#29: Disjoint Sets Intro Review for exam 3.

April 2, 2018 · *Wade Fagen-Ulmschneider*

Building a Heap with an Array of Data

- Assumption: Data already exists as an unsorted array in memory.
- Goal: Organize the data as a minHeap as fast as possible.



-	B	U	I	L	D	H	E	A	P	N	O	W			

Solutions:

1. Sort the array, O(n lg(n))
2. Use Heap::insert for every element, O(n lg(n))
3. Use a heapifyDown strategy on half the array:

Heap.cpp (partial)

```
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = _parent(size_); i > 0; i--) {
4      heapifyDown(i);
5    }
6  }
```

Theorem: The running time of buildHeap on array of size n is:

_____.

Strategy:

Define S(h):
Let **S(h)** denote the sum of the heights of all nodes in a complete tree of height **h**.

S(0) = [ ] 0

S(1) = [ ] [ ] 1

S(2) =

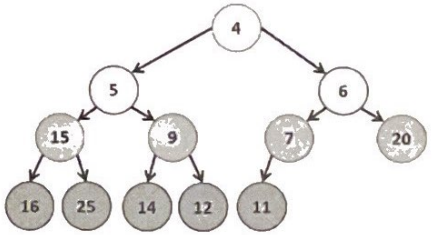
S(h) = $2^{h+1} - h - 2$.

Proof of S(h) by Induction:

Finally, finding the running time:

$S(h) \sim O(h 2^h) = O(2^{\log(n)}) = O(n)$.

Heap Sort



Algorithm:

1. buildHeap()
2. removeMin n times, place min at end.
3. reverse the order, use [0] as our start.

Running time?

$O(n) + n * O(\log n) + O(n) = O(n \lg n)$.

Why do we care about another sort?

in memory stable sort.

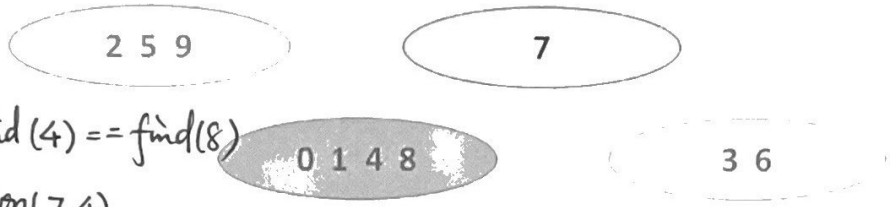
Disjoint Sets

Let **R** be an equivalence relation on *us* where **(s, t) ∈ R** if **s** and **t** have the same favorite among:

{ ____, pubG, FW, Zelda ____, ____ }

P.P M.S F.M D.B P.S Y.X

Examples:



find(4) == find(8)

union(7,4)

Building Disjoint Sets:

- Maintain a collection S = {s_0, s_1, ... s_k}
- Each set has a representative member.
- ADT:

  void makeSet(const T & t);
  void union(const T & k1, const T & k2);
  T & find(const T & k);

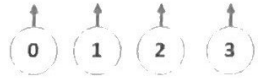
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Operation: find(k)

Operation: union(k1, k2)

Implementation #2:

- Continue to use an array where the index is the key
- The value of the array is:
  - **-1**, if we have found the representative element
  - **The index of the parent**, if we haven't found the rep. element



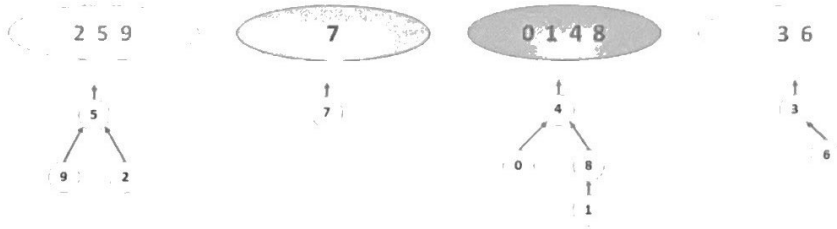
[0]	[1]	[2]	[3]

[0]	[1]	[2]	[3]

[0]	[1]	[2]	[3]

[0]	[1]	[2]	[3]

Example:



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
4	8	5	6	-1	-1	-1	-1	4	5

...where is the error in this table?

CS 225 – Things To Be Doing:

1. MP5 deadline tonight Monday, April 2nd
2. Theory Exam 3 starts tomorrow (Tuesday, April 3rd)
3. lab_heap starts on Wednesday
4. Daily POTDs are ongoing!

CS 225

#30: Disjoint Sets

April 4, 2018 · *Wade Fagen-Ulmschneider*

Disjoint Sets

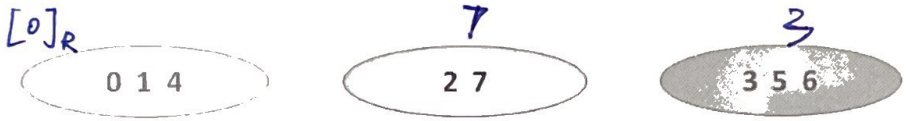
Let **R** be an equivalence relation. We represent R as several disjoint sets. Two key ideas from Monday:

- Each element exists in exactly one set.
- Every set is an equitant representation.
  - Mathematically: $4 \in [0]_R \rightarrow 8 \in [0]_R$
  - Programmatically: find(4) == find(8)

Building Disjoint Sets:

- Maintain a collection $S = \{s_0, s_1, \ldots s_k\}$
- Each set has a representative member.
- ADT:

```
void makeSet(const T & t);
void union(const T & k1, const T & k2);
T & find(const T & k);
```



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	0	7	3	0	3	3	7		

Operation: find(k) O(1)

match the elements with the set index.

Operation: union(k1, k2) O(n)

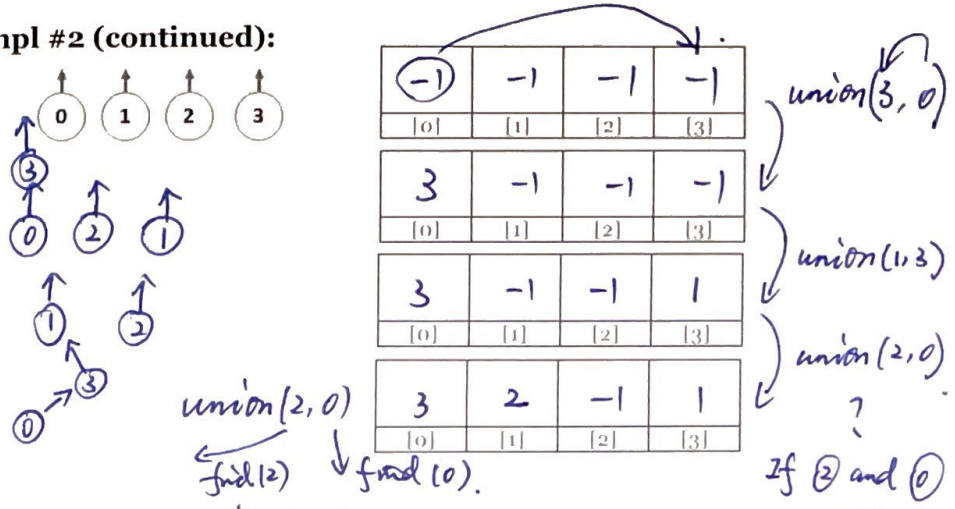
needs to go through the entire set, what it is ?

Implementation #2:

- Continue to use an array where the index is the key
- The value of the array is:
  - **-1**, if we have found the representative element
  - **The index of the parent**, if we haven't found the rep. element

upTree.

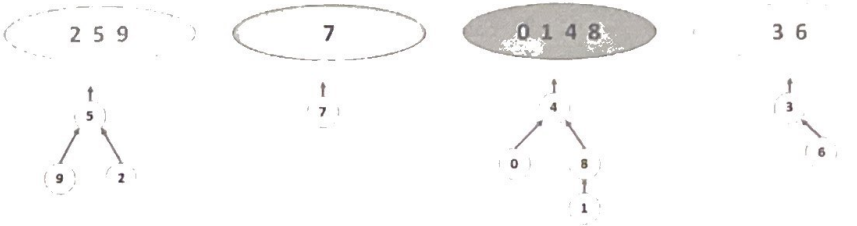
Impl #2 (continued):



[0]	[1]	[2]	[3]
-1	-1	-1	-1
3	-1	-1	-1
3	-1	-1	1
3	2	-1	1

Example:

union (2, 1)



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
4	8	5	6	-1	-1	-1	-1	4	5

...where is the error in this table?

The worst uptree : linked list.

Implementation – DisjointSets::find

DisjointSets.cpp (partial)

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else { return _find( s[i] ); }
4  }
```

What is the running time of `find`?

O(n) = O(h).

What is the ideal UpTree?

one root, a lot of children.

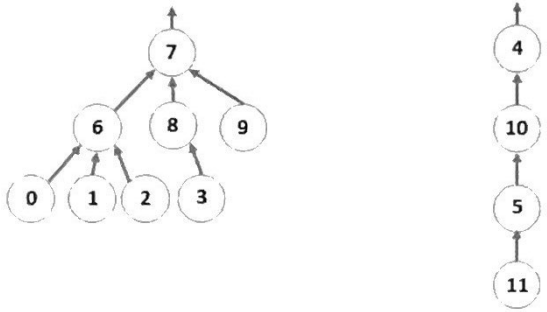
Implementation – DisjointSets::union

DisjointSets.cpp (partial)

```
1  void DisjointSets::union(int r1, int r2) {
2
3
4  }
```

How do we want to union the two UpTrees?

Building a Smart Union Function



The implementation of this visual model is the following:

6	6	6	8	-1	10	7	-1	7	7	4	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

What are possible strategies to employ when building a "smart union"?

Smart Union Strategy #1: _____
Idea: Keep the height of the tree as small as possible!

Metadata at Root:

After `union( 4, 7 )`:

6	6	6	8		10	7		7	7	4	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Smart Union Strategy #2: _____
Idea: Minimize the number of nodes that increase in height.
(Observe that the tree we union have all their nodes gain in height.)

Metadata at Root:

After `union( 4, 7 )`:

6	6	6	8		10	7		7	7	4	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Smart Union Implementation:

DisjointSets.cpp (partial)

```
1  void DisjointSets::unionBySize(int root1, int root2) {
2    int newSize = arr_[root1] + arr_[root2];
3
4    if ( arr_[root1] < arr_[root2] ) {
5      arr_[root2] = root1;  arr_[root1] = newSize;
6    } else {
7      arr_[root1] = root2;  arr_[root2] = newSize;
8    }
9  }
```

CS 225 – Things To Be Doing:

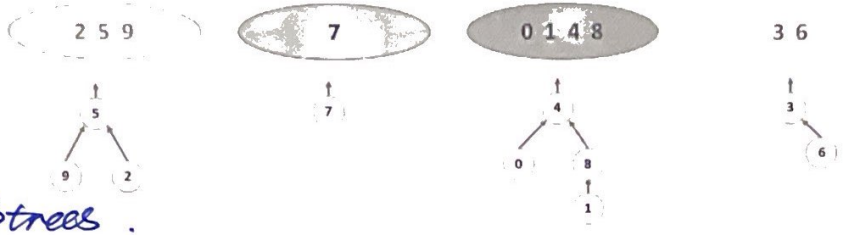
1. Theory Exam 3 is on-going
2. MP6 released; Extra Credit deadline on Monday, April 9th
3. lab_heaps released today
4. Daily POTDs are ongoing!

CS 225

#31: Disjoint Sets Implementation

April 6, 2018 · *Wade Fagen-Ulmschneider*

Disjoint Sets



uptrees.

4	8	5	-1	-1	-1	3	-1	4	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

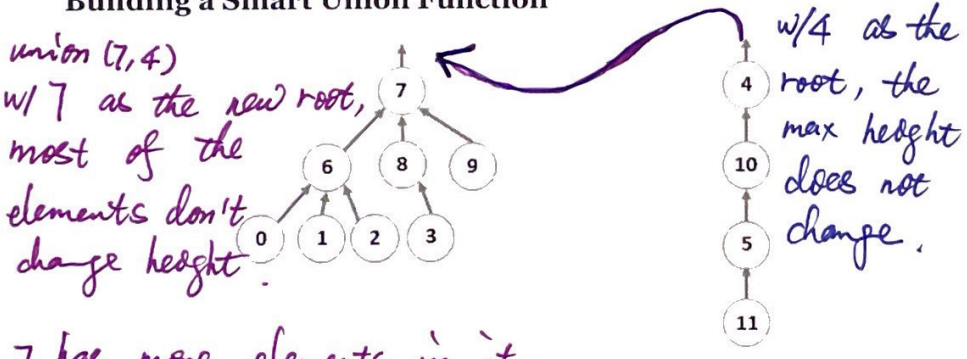
Implementation – DisjointSets::union

DisjointSets.cpp (partial)

```
1  void DisjointSets::union(int r1, int r2) {
2  // assume that r1 and r2 are roots.
3
4  }
```

How do we want to union the two UpTrees?

Building a Smart Union Function



The implementation of this visual model is the following:

6	6	6	8	-1	10	7	-1	7	7	4	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

What are possible strategies to employ when building a "smart union"?

Smart Union Strategy #1: _____

Idea: Keep the height of the tree as small as possible!

Metadata at Root:

After `union( 4, 7 )`:

6	6	6	8	-3	10	7	-2	7	7	4	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Smart Union Strategy #2: union by ~~idea~~ size.

Idea: Minimize the number of nodes that increase in height.

(Observe that the tree we union have all their nodes gain in height.)

average better performance.

Metadata at Root:

After `union( 4, 7 )`:

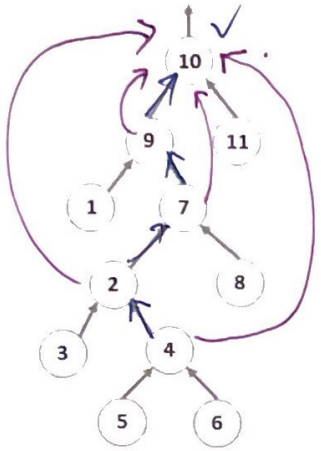
6	6	6	8	-4	10	7	-8	7	7	4	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Smart Union Implementation:

DisjointSets.cpp (partial)

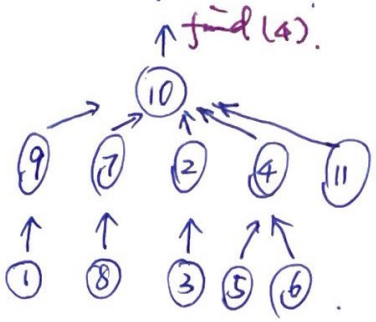
```
1  void DisjointSets::unionBySize(int root1, int root2) {
2    int newSize = arr_[root1] + arr_[root2];
3
4    if ( arr_[root1] < arr_[root2] ) {
5      arr_[root2] = root1;  arr_[root1] = newSize;
6    } else {
7      arr_[root1] = root2;  arr_[root2] = newSize;
8    }
9  }
```

Path Compression:



bad uptree : find (4) goes up

4 → 2 → 7 → 9 → 10



A Review of Major Data Structures so Far

Array-based	List/Pointer-based
- Sorted Array	- Singly Linked List
- Unsorted Array	- Doubly Linked List
- Stacks	- Skip Lists
- Queues	- Trees
- Hashing	- BTree
- Heaps	- Binary Tree
- Priority Queues	- Huffman Encoding
- UpTrees	- kd-Tree
- Disjoint Sets	- AVL Tree

UpTree Implementation with a smart union function and path compression:

DisjointSets.cpp (partial)

```
1  int DisjointSets::find(int i) {
2    if ( arr_[i] < 0 ) { return i; }
3    else { return                    find( arr_[i] ); }
4  }
```

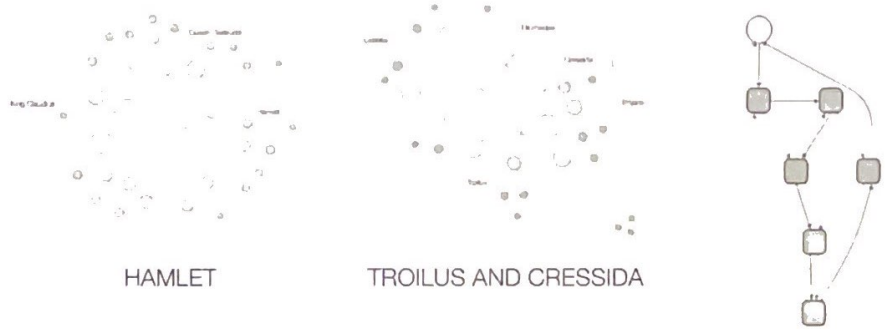
lg(n).

DisjointSets.cpp (partial)

```
1  void DisjointSets::unionBySize(int root1, int root2) {
2    int newSize = arr_[root1] + arr_[root2];
3
4    // If arr_[root1] is less than (more negative), it is the
5    // larger set; we union the smaller set, root2, with root1.
6    if ( arr_[root1] < arr_[root2] ) {
7      arr_[root2] = root1;
8      arr_[root1] = newSize;
9    }
10
11   // Otherwise, do the opposite:
12   else {
13     arr_[root1] = root2;
14     arr_[root2] = newSize;
15   }
16 }
```

O(1) given root index.

An Introduction to Graphs



得先.

CS 225 – Things To Be Doing:
1. Theory Exam 3 final day is **today** 2. lab_heaps due Sunday, April 8th 3. MP6 released; Extra Credit deadline on Monday, April 9th 4. Daily POTDs are ongoing!

iterated log function: # you can take a lot of numbers.

$$\log^*(n) = \begin{cases} 0, & n \le 1 \\ 1 + \log^*(\log(n)), & n > 1 \end{cases}$$

what is $\lg^*(2^{65536}) = 5$.

65536 → 16 → 4 → 2 → 1.

Last chapter

CS 225

#32: Graphs: An Introduction

April 9, 2018 · *Wade Fagen-Ulmschneider*

Disjoint Sets

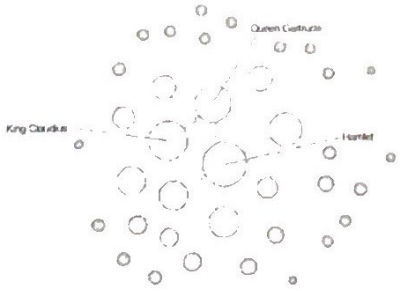
- Worst case running time of find(k):
- Worst case running time of union(r1, r2), given roots:
- Iterated log: **log*(n)** = *number of times you can take a log*
- Overall running time:
  - A total of **m** union/find operation runs in:

感觉人生

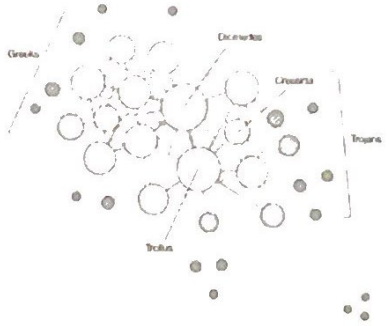
A Review of Major Data Structures so Far

Array-based	List/Pointer-based
- Sorted Array	- Singly Linked List
- Unsorted Array	- Doubly Linked List
- Stacks	- Skip Lists
- Queues	- Trees
- Hashing	- BTree
- Heaps	- Binary Tree
- Priority Queues	- Huffman Encoding
- UpTrees	- kd-Tree
- Disjoint Sets	- AVL Tree

An Introduction to Graphs

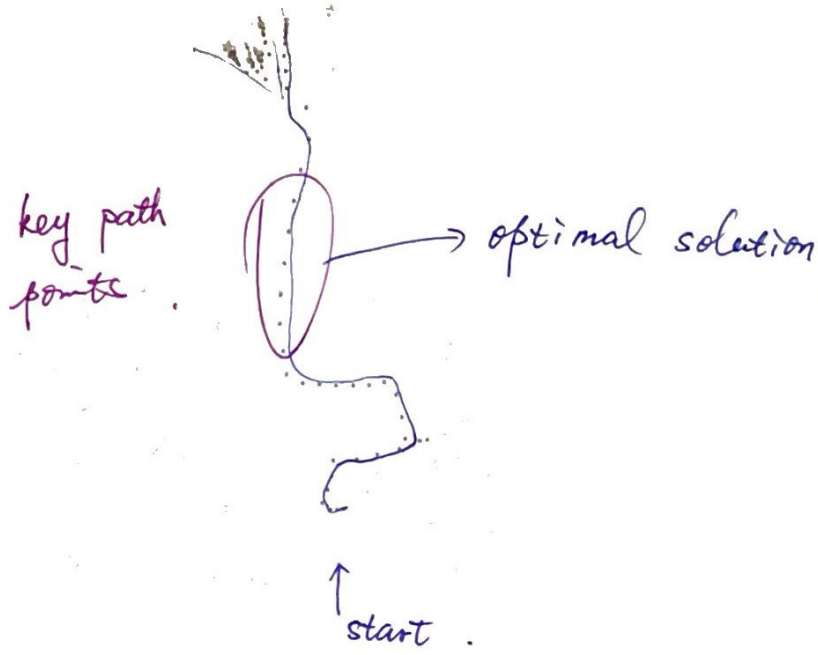


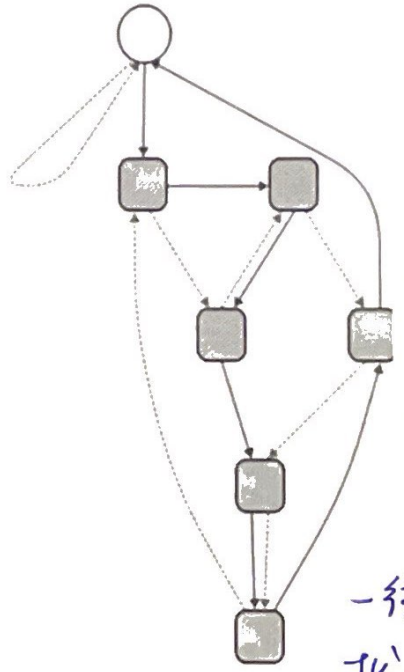
HAMLET



TROILUS AND CRESSIDA

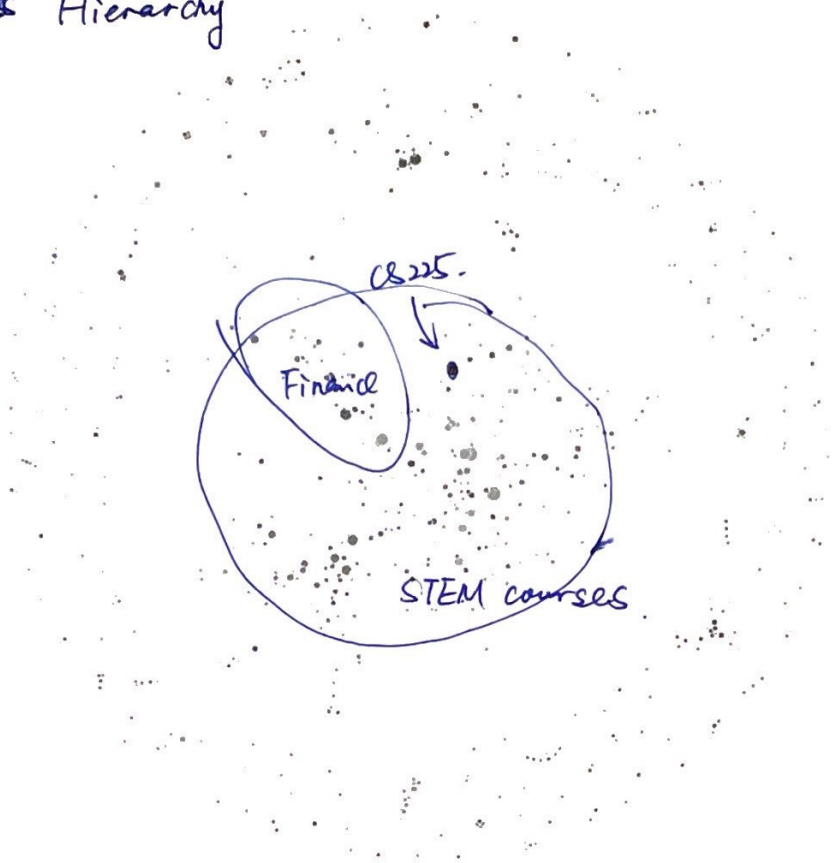
Martin Grandjean. 2016 NLP corpus relation.





一往无前危山行
拨开云雾见光明
梦里花开
幻 真歌

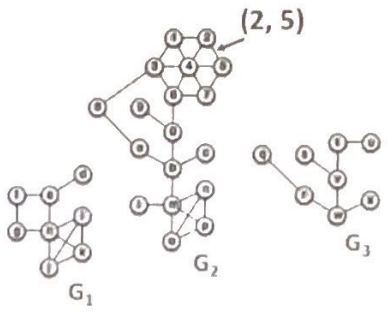
Class Hierarchy



Motivation:
Graphs are awesome data structures that allow us to represent an enormous range of problems. To study these problems, we need:

1. A common vocabulary to talk about graphs
2. Implementation(s) of a graph
3. Traversals on graphs
4. Algorithms on graphs

Graph Vocabulary
Consider a graph **G** with vertices **V** and edges **E**, **G=(V,E)**.



Incident Edges:
I(v) = { (x, v) in E }

Degree(v): **|I|**

Adjacent Vertices:
A(v) = { x : (x, v) in E }

Path(G_2): Sequence of vertices connected by edges

Cycle(G_1): Path with a common begin and end vertex.

Simple Graph(G): A graph with no self loops or multi-edges.

Subgraph(G): **G' = (V', E')**:
V' ∈ V, E' ∈ E, and (u, v) ∈ E → u ∈ V', v ∈ V'

Graphs that we will study this semester include:
Complete subgraph(G)
Connected subgraph(G)
Connected component(G)
Acyclic subgraph(G)
Spanning tree(G)

CS 225 – Things To Be Doing:

1. Theory Exam 3 final day is **today**
2. lab_heaps due Sunday, April 8th
3. MP6 released; Extra Credit deadline on Monday, April 9th
4. Daily POTDs are ongoing!

CS 225

#33: Graph Vocabulary + Implementation

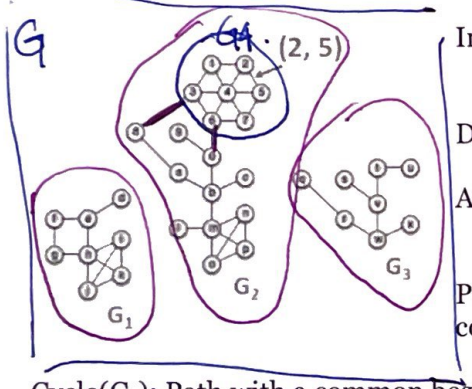
April 11, 2018 · *Wade Fagen-Ulmschneider*

Motivation:
Graphs are awesome data structures that allow us to represent an enormous range of problems. To study these problems, we need:

1. A common vocabulary to talk about graphs
2. Implementation(s) of a graph
3. Traversals on graphs
4. Algorithms on graphs

Graph Vocabulary

Consider a graph **G** with vertices **V** and edges **E**, **G=(V,E)**.



Incident Edges:
I(v) = { (x, v) in E }

Degree(v): **|I|**

Adjacent Vertices:
A(v) = { x : (x, v) in E }

Path(G_2): Sequence of vertices connected by edges

Cycle(G_1): Path with a common begin and end vertex.

Simple Graph(G): A graph with no self loops or multi-edges.

Subgraph(G): **G' = (V', E')**: G1, G2, G3.
V' ∈ V, E' ∈ E, and (u, v) ∈ E → u ∈ V', v ∈ V'

all vertices of the edges are part of the vertices of that subgraph set.

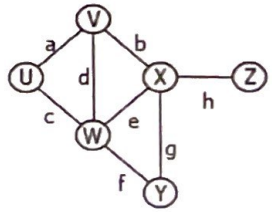
~~edges be named after vertices~~

Graphs that we will study this semester include:
- Complete subgraph(G)
- Connected subgraph(G)
- Connected component(G)
- Acyclic subgraph(G)
- Spanning tree(G)

Size and Running Times

Running times are often reported by **n**, the number of vertices, but often depend on **m**, the number of edges.

For arbitrary graphs, the minimum number of edges given a graph that is:



Not Connected: 0 $|E| = \phi$

Minimally Connected:* $m = n - 1$.

The maximum number of edges given a graph that is:

Simple:

n		m
1	•	0
2	—	1 (+1)
3	△	3 (+2)
4	⊠	6 (+3)
n		$m = \sum_{k=0}^{n-1} k$

$m \sim n^2$.

Not Simple: infinite ← multi-edges. $m = \infty$

The relationship between the degree of the graph and the edges:

$\sum deg(v) = 2m$.

Proving the Size of a Minimally Connected Graph

Theorem: Every minimally connected graph **G=(V, E)** has **|V|-1** edges.

Proof of Theorem

Consider an arbitrary, minimally connected graph **G=(V, E)**.

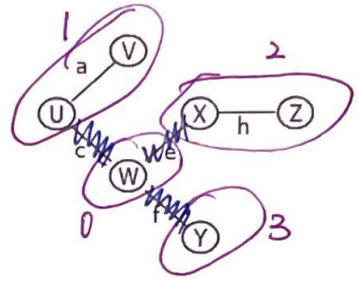
Lemma 1: Every connected subgraph of **G** is minimally connected.
(Easy proof by contradiction left for you.)

Inductive Hypothesis: For any **j** < |V|, any minimally connected graph of **j** vertices has **j-1** edges.

Suppose |V| = 1:
Definition: A minimally connected graph of 1 vertex has 0 edges.
Theorem: |V|-1 edges ➔ 1-1 = 0.

Suppose |V| > 1:
Choose any vertex **u** and let **d** denote the degree of **u**.

Remove the incident edges of **u**, partitioning the graph into 4 components: $C_0 = (V_0, E_0)$, ..., $C_d = (V_d, E_d)$.



By Lemma 1, every component C_k is a minimally connected subgraph of **G**.

By our induction hypothesis:

$|E_k| = |V_k| - 1$.

Finally, we count edges:

$$|E| = d + \sum_{k=0}^{d} (|V_k| - 1) = n - 1.$$

Graph ADT

Data	Functions

Vertices
 	
Edges
 	
Some data structure maintaining the structure between vertices and edges.
	`insertVertex(K key);`
	`insertEdge(Vertex v1, Vertex v2, K key);`
	`removeVertex(Vertex v);`
	`removeEdge(Vertex v1, Vertex v2);`
	`incidentEdges(Vertex v);`
	`areAdjacent(Vertex v1, Vertex v2);`
	`origin(Edge e);`
	`destination(Edge e);` } directed graph.

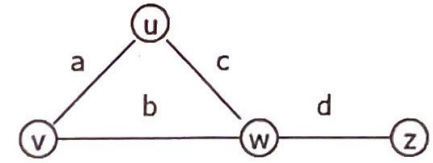
Hash Table. Linked List.

Graph Implementation #1: Edge List

addEdge (v_1, v_2, k)

Vert.	Edges
u v w z	u v a v w b u w c w z d

n m



Operations:

insertVertex(K key): O(1) hash table.

★ removeVertex(Vertex v): O(1) + O(m) = O(m)
remove from vertex remove edges.

areAdjacent(Vertex v1, Vertex v2):

incidentEdges(Vertex v):

O(m) → $O(n^2)$. ⇐ O(1).

O(m) → $O(n^2)$.

Graph Implementation #2: Adjacency Matrix

but bcz the matrix may take time.

Vert.	Edges	Adj. Matrix
u v w z	u v a v w b u w c w z d	u: — 1 1 0 v: — 1 0 w: — 1 z: —

undirected graph, only upper

CS 225 – Things To Be Doing:
1. Topic list for Programming Exam C available; starts Tuesday 4/17 2. lab_puzzles released today 3. MP6 released due on Monday, April 16th 4. Daily POTDs are ongoing!

CS 225

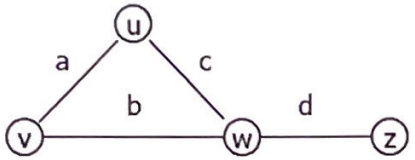
#34: Graph Implementation

April 13 2018 · *Wade Fagen-Ulmschneider*

Graph Implementation #1: Edge List

- HashTable storage of our vertex set
- List storage of our edge set
- O(1) runtime: insertVertex
- O(m) runtime: removeVertex, areAdjacent, and incidentEdges

Graph Implementation #2: Adjacency Matrix



Vert.
u
v
w
z

Edges		
u	v	a
v	w	b
u	w	c
w	z	d

Adj. Matrix

	u	v	w	z
u				0
v				0
w				
z				

$\frac{O(n^2)}{O(n)} = O(n)^*$

$O(n^2)$ double every $O(n)$ times.

double the matrix $m \times n \to 2m \times 2n$

n^2 overhead.

Operations on an Adjacency Matrix:

insertVertex(K key): $O(1) \to O(1) + O(n)^* = O(n)^*$

$\frac{O(n^2)}{O(n)} = O^*(n)$.

we don't know whether m or n is bigger.

removeVertex(Vertex v): $O(m) \to O(1) + O(n)$.

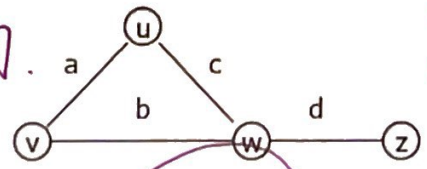
areAdjacent(Vertex v1, Vertex v2): $O(m) \to O(1)$.

incidentEdges(Vertex v): $O(m) \to O(n)$.

Graph Implementation #3: Adjacency List

用空间换时间.

7 pointers to maintain the incident edges for u.



Vertex List
u
v
w
z

Edges		
u	v	a
v	w	b
u	w	c
w	z	d

hash table

linked list

$\sum$ nodes $= 2m$.

Operations on an Adjacency List:

insertVertex(K key): $O(1)$

removeVertex(Vertex v): $O(deg(u)) \to$ pretty much optimal time.

areAdjacent(Vertex v1, Vertex v2): $\min(O(deg(v_1)), O(deg(v_2)))$

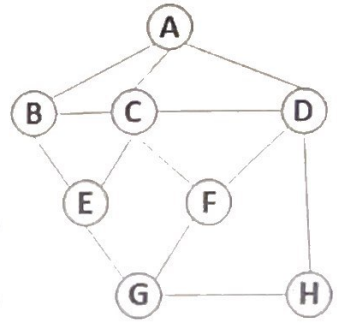
incidentEdges(Vertex v): $O(deg(v))$ size of linked list.

It matters for graph traversal, etc

Running Times of Classical Graph Implementations

	Edge List	Adj. Matrix	Adj. List
Space	n+m	~~n+m~~ n^2	~~n^2~~ n+m
insertVertex	1	n	1
removeVertex	m	n	deg(v)
insertEdge	1	1	1
removeEdge	1	1	1
incidentEdges	m	n	deg(v)
areAdjacent	m	1	min(deg(v), deg(w))

BST Graph Traversal



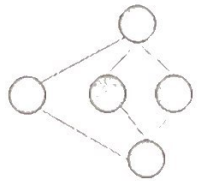
How do the algorithms compare?

...is one always better?

Graph Traversal

Objective: Visit every vertex and every edge in the graph.
Purpose: Search for interesting sub-structures in the graph.

We've seen traversal before – this is only slightly different:

BST	Graph
	

CS 225 – Things To Be Doing:
1. Topic list for Programming Exam C available; starts Tuesday 4/17 2. lab_puzzles ongoing; due Sunday, April 15th 3. MP6 due on Monday, April 16th 4. Daily POTDs are ongoing!

user case #1: sparse graph, $m < n$, not fully connected.

CS 225

#35: Graph Traversals

April 16, 2018 · *Wade Fagen-Ulmschneider*

$\deg(v) < n$ → adjacency list

Running Times of Classical Graph Implementations

	Edge List	Adj. Matrix	Adj. List
Space	n+m	n^2	n+m
insertVertex	1	n	1
removeVertex	m	n	deg(v)
insertEdge	1	1	1
removeEdge	1	1	1
incidentEdges	m	n	deg(v)
areAdjacent	m	1	min(deg(v), deg(w))

Implementations and Use Cases

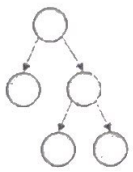
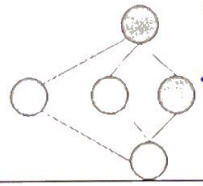
Ex. 1: sparse graph, $m < n$, $\deg(v) < n$. → adjacency list.

Ex. 2: dense, graph, $m \sim n^2$, $\deg(v) \sim n$, → depends on use case: run areAdjacent ?

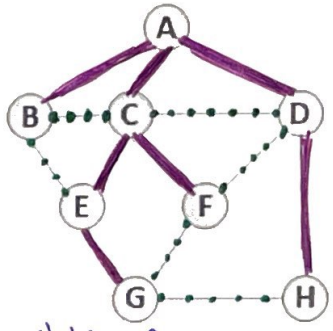
Graph Traversal

Objective: Visit every vertex and every edge in the graph.
Purpose: Search for interesting sub-structures in the graph.

We've seen traversal before – this is different:

BST	Graph
	
△ ordered △ obvious start (root) △ notion of completeness.	△ Any order △ arbitrary start point △ No clean notion of completeness.

BFS Graph Traversal



BFS. visit adjacent nodes first.

edge labels: ① discovery ② cross ...

n	d	v	p	list of adjacent nodes
A	0	✓	–	C B D
B	1	✓	A	A C E
C	1	✓	A	B A D E F
D	1	✓	A	A C F H
E	2	✓	C	B C G
F	2	✓	C	C D G
G				E F H
H				D G.

(distance, visited, predecessor; who discovers these edges)

queue: G H F E D ~~B~~ ~~C~~ ~~A~~ (B, C, A)

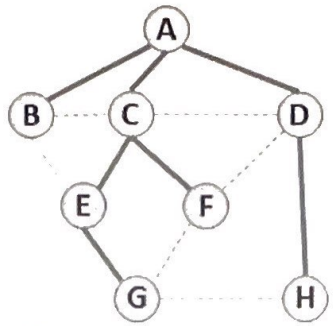
Pseudocode for BFS

```
1  BFS(G):
2    Input: Graph, G
3    Output: A labeling of the edges on
4        G as discovery and cross edges
5
6    foreach (Vertex v : G.vertices()):
7      setLabel(v, UNEXPLORED)
8    foreach (Edge e : G.edges()):
9      setLabel(e, UNEXPLORED)
10   foreach (Vertex v : G.vertices()):
11     if getLabel(v) == UNEXPLORED:
12       BFS(G, v)
13
14 BFS(G, v):
15   Queue q
16   setLabel(v, VISITED)
17   q.enqueue(v)
18
19   while !q.empty():          O(n)
20     v = q.dequeue()          deg(v).
21     foreach (Vertex w : G.adjacent(v)):
22       if getLabel(w) == UNEXPLORED:
23         setLabel(v, w, DISCOVERY)
24         setLabel(w, VISITED)
25         q.enqueue(w)
26       elseif getLabel(v, w) == UNEXPLORED:
27         setLabel(v, w, CROSS)
```

while !q.empty()
1. dequeue
2. report it
3. loop through the adjacent vertices
   - update edge (discovery / cross)
   - mark as visited.
   - enqueue.

will discover all vertices in a connected part of graph.

BST Graph Observations



1. Does our implementation handle disjoint graphs? How?

   Yes. pseudo-code 10 – 12

   a. How can we modify our code to count components?

   after line 12, increase # of components.

2. Can our implementation detect a cycle? How?

   Yes.

   a. How can we modify our code to store update a private member variable `cycleDetected_`?

   If Line 27 runs, we have a cross edge, implying a cycle.

3. What is the running time of our algorithm?

   $\sum^{n} deg(v) = 2m$ $\quad O(n) + O(m) = O(n+m)$ with an adjacency list.

4. What is the shortest path between **A** and **H**?

5. What is the shortest path between **E** and **H**?

   a. What does that tell us about BFS?

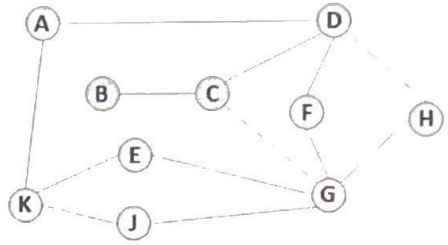
6. What does a cross edge tell us about its endpoints?

7. What structure is made from discovery edges in **G**?

Big Ideas: Utility of a BFS Traversal

Obs. 1: Traversals can be used to count components.
Obs. 2: Traversals can be used to detect cycles.
Obs. 3: In BFS, **d** provides the shortest distance to every vertex.
Obs. 4: In BFS, the endpoints of a cross edge never differ in distance, d, by more than 1: $|\mathbf{d(u) - d(v)}| = 1$

Depth First Search – A Modification to BFS



Two types of edges: 1.

2.

Running Time of DFS:

Labeling:
- Vertex:
- Edge:

Queries:
- Vertex:
- Edge:

CS 225 – Things To Be Doing:

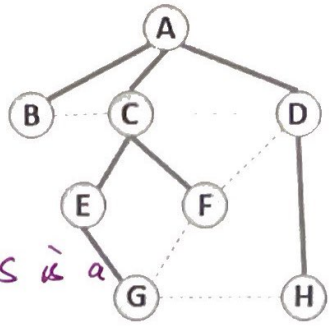
1. Programming Exam C starts Tuesday 4/17
2. MP6 due tonight, Monday, April 16th
3. lab_graphs available Wednesday
4. Daily POTDs are ongoing!

CS 225

#36: Minimum Spanning Trees

April 18, 2018 · *Wade Fagen-Ulmschneider*

BFS Graph Observations



1. Does our implementation handle disjoint graphs? How?

   yes.

   a. How can we modify our code to count components?

   every call to the inner-BFS is a graph component.

2. Can our implementation detect a cycle? How?

   detects a cycle at a cross edge.

   a. How can we modify our code to store update a private member variable `cycleDetected_`?

3. What is the running time of our algorithm?

   $O(m+n)$.

4. What is the shortest path between **A** and **H**?

   2, from our BFS data structure. we have a "distance". look at the predecessor of H.

5. What is the shortest path between **E** and **H**?

   A → D → H.

   a. What does that tell us about BFS?

   we don't know. Shortest path can only be found at the starting point.

6. What does a cross edge tell us about its endpoints?

   the change of magnitude of d at a cross is small. $|\Delta d|$ at a cross edge ≤ 1.

7. What structure is made from discovery edges in **G**?

   spanning tree, a tree that connect all vertices.

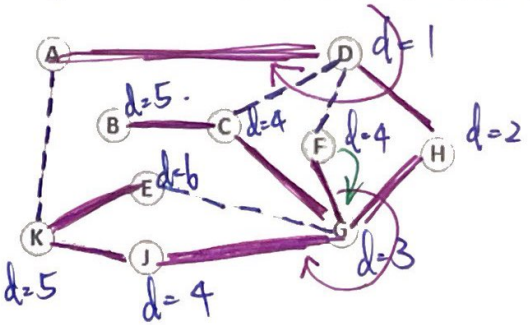
Pseudocode for DFS

```
1  BFS(G):
2    Input: Graph, G
3    Output: A labeling of the edges on
4        G as discovery and ~~cross~~ back edges
5
6    foreach (Vertex v : G.vertices()):
7      setLabel(v, UNEXPLORED)
8    foreach (Edge e : G.edges()):
9      setLabel(e, UNEXPLORED)
10   foreach (Vertex v : G.vertices()):
11     if getLabel(v) == UNEXPLORED:
12       BFS(G, v)
13
14 BFS(G, v):
15   ~~Queue q~~
16   setLabel(v, VISITED)
17   ~~q.enqueue(v)~~
18
19   ~~while !q.empty():~~
20     ~~v = q.dequeue()~~
21     foreach (Vertex w : G.adjacent(v)):
22       if getLabel(w) == UNEXPLORED:
23         setLabel(v, w, DISCOVERY)
24         setLabel(w, VISITED)
25         ~~q.enqueue(w)~~ DFS(G, w).
26       elseif getLabel(v, w) == UNEXPLORED:
27         setLabel(v, w, ~~CROSS~~ BACK.
```

Big Ideas: Utility of a BFS Traversal

Obs. 1: Traversals can be used to count components.
Obs. 2: Traversals can be used to detect cycles.
Obs. 3: In BFS, **d** provides the shortest distance to every vertex.
Obs. 4: In BFS, the endpoints of a cross edge never differ in distance, d, by more than 1: $|\mathbf{d(u) - d(v)}| = 1$

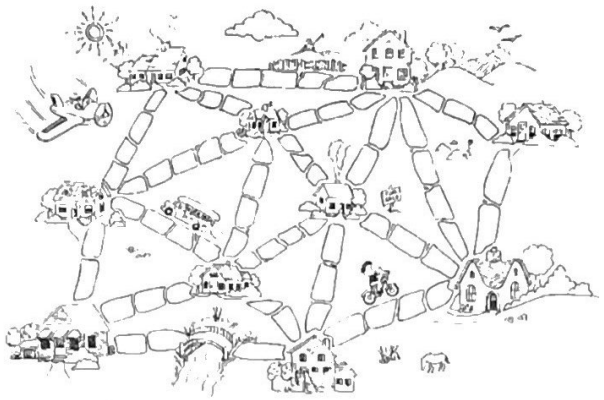
Depth First Search – A Modification to BFS



Discovery

..... Back.

back edge. leads us closer to the beginning, the distance back is unbounded.



"The Muddy City" by CS Unplugged, **Creative Commons BY-NC-SA 4.0**

A **Spanning Tree** on a connected graph **G** is a subgraph, **G'**, such that:

1. Every vertex in G is in G' and
2. G' is connected with the minimum number of edges

This construction will always create a new graph that is a tree (connected, acyclic graph) that spans G.

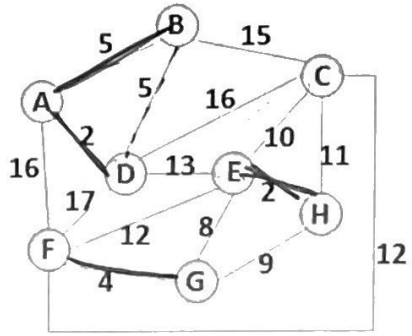
A **Minimum Spanning Tree** is a spanning tree with the minimal total edge weights among all spanning trees.

- Every edge must have a weight
  - The weights are unconstrained, except they must be additive *(eg: can be negative, can be non-integers)*
- Output of a MST algorithm produces G':
  - G' is a spanning graph of G
  - G' is a tree
  - G' has a minimal total weight among all spanning trees

CS 225 – Things To Be Doing:

1. Programming Exam C ongoing
2. MP7 is released; EC due tonight, Monday, April 23th
3. lab_graphs available today; dues Sunday, April 22nd
4. Daily POTDs are ongoing!

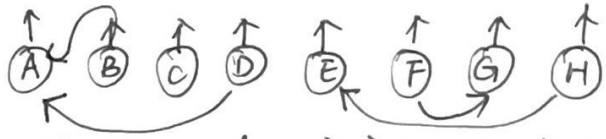
Kruskal's Algorithm



min heap as edges.

~~(A, D)~~	2
~~(E, H)~~	2
~~(F, G)~~	4
~~(A, B)~~	5
~~(B, D)~~	5
(G, E)	
(G, H)	
(E, C)	
(C, H)	
(E, F)	
(F, C)	
(D, E)	
(B, C)	
(C, D)	
(A, F)	
(D, F)	

the same equivalent set.



pop A, D they joint disjoint set, mark AD.

Pseudocode for Kruskal's MST Algorithm

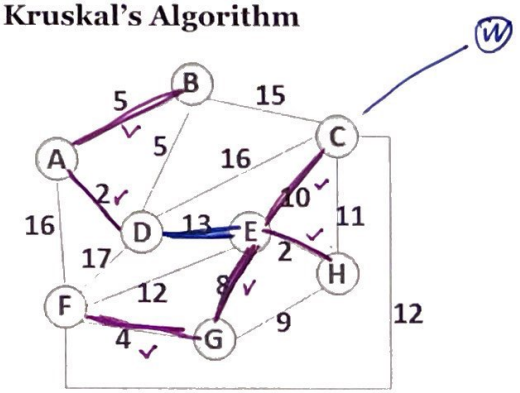
```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G):
4      forest.makeSet(v)
5
6    PriorityQueue Q     // min edge weight
7    foreach (Edge e : G):
8      Q.insert(e)
9
10   Graph T = (V, {})
11
12   while |T.edges()| < n-1:
13     Vertex (u, v) = Q.removeMin()
14     if forest.find(u) != forest.find(v):
15        T.addEdge(u, v)
16        forest.union( forest.find(u),
17                      forest.find(v) )
18
19   return T
```

CS 225

#37: Kruskal + Prim's Algorithm

April 20, 2018 · *Wade Fagen-Ulmschneider*

Kruskal's Algorithm

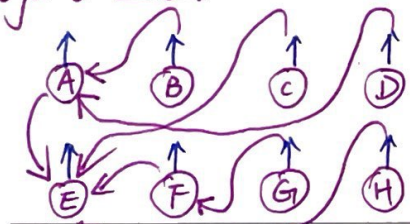


priority queue

edge set

(A, D) ✓
(E, H) ✓
(F, G) ✓
(A, B) ✓
(B, D) ✗
(G, E) ✓
(G, H) ✗
(E, C) ✓
(C, H) ✗
(E, F) ✗
(F, C) ✗
(D, E) ✓
(B, C)
(C, D)
(A, F)
(D, F)

Disjoint Sets




```
   Pseudocode for Kruskal's MST Algorithm
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G):
4      forest.makeSet(v)
5
6    PriorityQueue Q     // min edge weight
7    foreach (Edge e : G):
8      Q.insert(e)
9
10   Graph T = (V, {})
11
12   while |T.edges()| < n-1:
13     Vertex (u, v) = Q.removeMin()
14     if forest.find(u) != forest.find(v):
15       T.addEdge(u, v)
16       forest.union( forest.find(u),
17                     forest.find(v) )
18
19   return T
```

} O(n).

} heap O(m) sorted array O(m log m)

O(n).

O(m) ← heap O(log n) s.array O(1).

} O(1).

$m = O(n^2) \leq c \cdot n^2$.

log is an increasing function. $\lg(m) \to O(\lg(n))$

$\log(m) \leq \log(c \cdot n^2)$

$= \log c + 2\log n$

Kruskal's Running Time Analysis

We have multiple choices on which underlying data structure to use to build the Priority Queue used in Kruskal's Algorithm:

Priority Queue Implementations:	Heap	Sorted Array
Building :6-8	O(m)	O(m log m) → O(m lg n)
Each removeMin :13	O(log m) → O(lg n)	O(1)*

Based on our algorithm choice: setup / build + iteration

Priority Queue Implementation:	Total Running Time
Heap 2: 10 → setup, 12: 18 → iter	O(n+m) + O(m lg n) = O(n+m lg n)
Sorted Array	O(n + m lg n) + O(m) = O(n + m lg n)

Reflections

Why would we prefer a Heap?

① A heap has a cheaper setup, useful if we stop early

② cheaper to update an edge weight in a heap

Why would be prefer a Sorted Array?

sorted arrays are extremely useful - may already have one. O(lg(n)) v.s. O(n)

→ 日常 ！

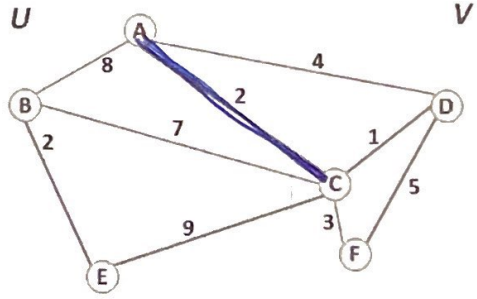
Partition Property

Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**.

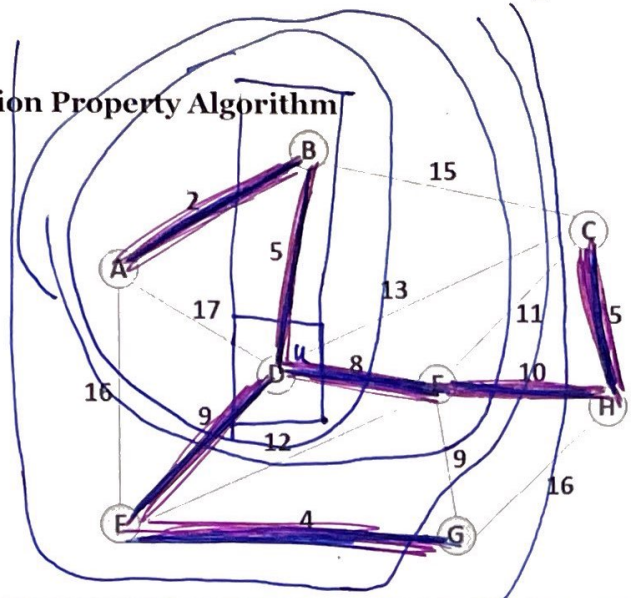
Let **e** be an edge of minimum weight across the partition.

Then **e** is part of some minimum spanning tree.

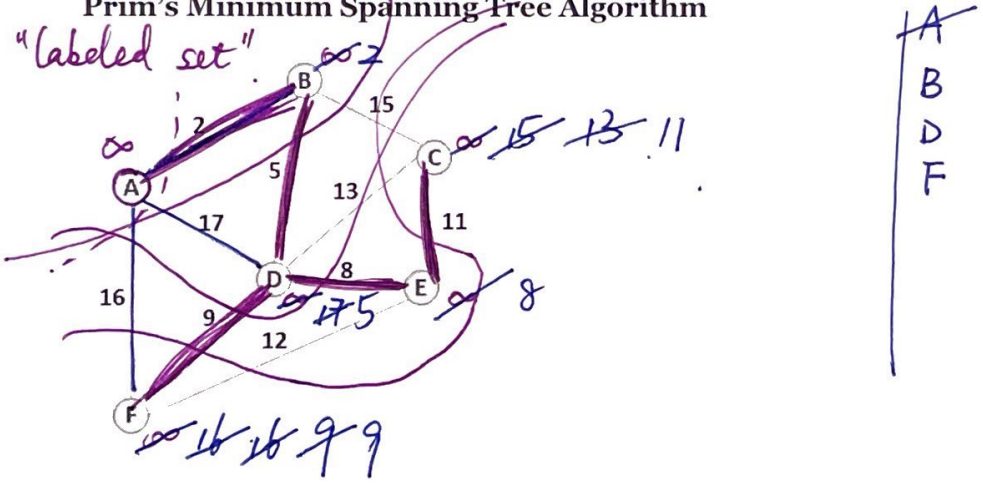
Proof in CS 374!



Partition Property Algorithm



Prim's Minimum Spanning Tree Algorithm



A
B
D
F

	Adj. Matrix	Adj. List
Heap		
Unsorted Array		

Pseudocode for Prim's MST Algorithm

```
1  PrimMST(G, s):
2    Input: G, Graph;
3           s, vertex in G, starting vertex of algorithm
4    Output: T, a minimum spanning tree (MST) of G
5
6    foreach (Vertex v : G):
7      d[v] = +inf
8      p[v] = NULL
9    d[s] = 0
10
11   PriorityQueue Q    // min distance, defined by d[v]
12   Q.buildHeap(G.vertices())
13   Graph T            // "labeled set"
14
15   repeat n times:
16     Vertex m = Q.removeMin()
17     T.add(m)
18     foreach (Vertex v : neighbors of m not in T):
19       if cost(v, m) < d[v]:
20         d[v] = cost(v, m)
21         p[v] = m
22
23   return T
```

parent pointer

Running Time of MST Algorithms

Kruskal's Algorithm:

Prim's Algorithm:

Q: What must be true about the connectivity of a graph when running an MST algorithm?

...what does this imply about the relationship between **n** and **m**?

CS 225 – Things To Be Doing:

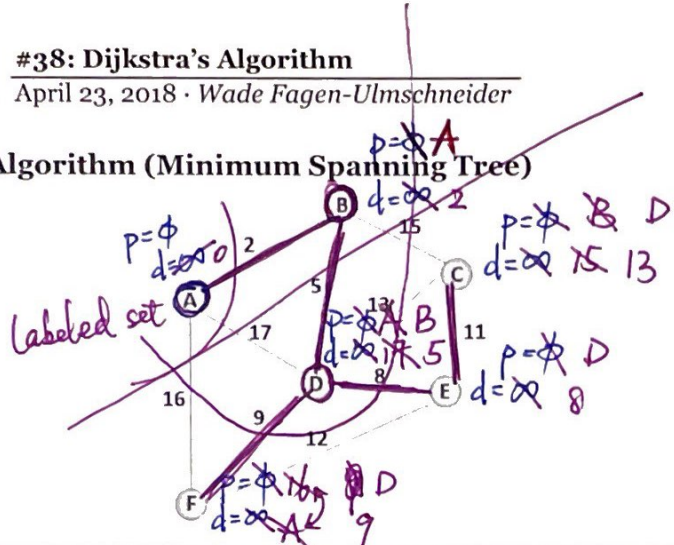
1. Programming Exam C ongoing
2. MP7 is released; EC due tonight, Monday, April 23th
3. lab_graphs available; due Sunday, April 22nd
4. Daily POTDs end **next** Friday (April 27th); +6 left to complete!

CS 225

#38: Dijkstra's Algorithm

April 23, 2018 · *Wade Fagen-Ulmschneider*

Prim's Algorithm (Minimum Spanning Tree)




```
Pseudocode for Prim's MST Algorithm
 1  PrimMST(G, s):
 2    Input: G, Graph;
 3           s, vertex in G, starting vertex of algorithm
 4    Output: T, a minimum spanning tree (MST) of G
 5
 6    foreach (Vertex v : G):
 7      d[v] = +inf
 8      p[v] = NULL
 9    d[s] = 0
10
11    PriorityQueue Q   // min distance, defined by d[v]
12    Q.buildHeap(G.vertices())
13    Graph T           // "labeled set"
14
15    repeat n times:
16      Vertex m = Q.removeMin()
17      T.add(m)
18      foreach (Vertex v : neighbors of m not in T):
19        if cost(v, m) < d[v]:
20          d[v] = cost(v, m)
21          p[v] = m
22
23    return T
```

	Adj. Matrix	Adj. List
Heap	$n^2 + m\lg(n)$	$n\lg(n) + m\lg(n)$
Unsorted Array	$n^2 + m = O(n^2)$	$n^2 + m = O(n^2)$

sparse graph : $n \sim m$ → heap with Adj. List

dense graph : $n^2 \sim m$ → unsorted array

Running Time of MST Algorithms

- Kruskal's Algorithm: $O(n + m\lg n)$ → n bound by m.
- Prim's Algorithm: $O(n\lg n + m\lg n)$.

Q: What must be true about the connectivity of a graph when running an MST algorithm?

The graph must be connected.

...what does this imply about the relationship between **n** and **m**?

$m \geq n-1$. → MST: $O(m\lg(n))$

~~$n\lg(n)+m$, Adj. List with a Fibonacci Heap~~

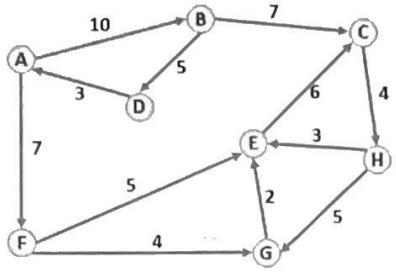
Q: Suppose we built a new heap that optimized the decrease-key operation, where decreasing the value of a key in a heap updates the heap in amortized constant time, or O(1)*. How does that change Prim's Algorithm runtime?

Fibonacci Heap : decrease key in $O(1)^*$ time.

d[v] = cost(r,m) → O(1)

Shortest Path Home:

Dijkstra's Algorithm (Single Source Shortest Path)

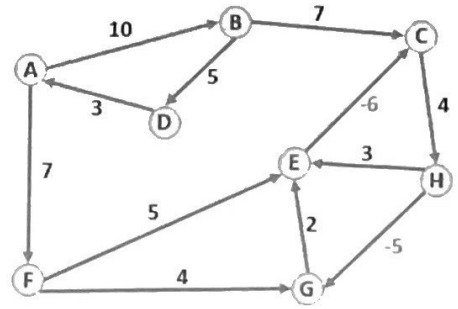


Dijkstra's Algorithm Overview:

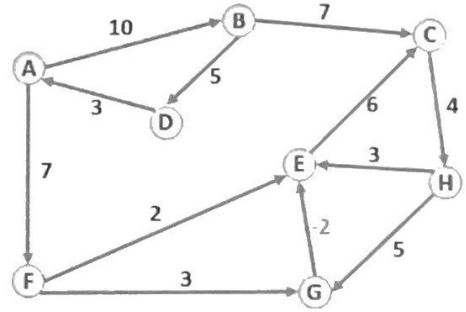
- The overall logic is the same as Prim's Algorithm
- We will modify the code in only two places – both involving the update to the distance metric.
- The result is a directed acyclic graph or DAG

	Pseudocode for Dijkstra's SSSP Algorithm
1	`DijkstraSSSP(G, s):`
2	`  Input: G, Graph;`
3	`  s, vertex in G, starting vertex of algorithm`
4	`  Output: T, DAG w/ shortest paths (and distances) to s`
5	
6	`  foreach (Vertex v : G):`
7	`  d[v] = +inf`
8	`  p[v] = NULL`
9	`  d[s] = 0`
10	
11	`  PriorityQueue Q  // min distance, defined by d[v]`
12	`  Q.buildHeap(G.vertices())`
13	`  Graph T  // "labeled set"`
14	
15	`  repeat n times:`
16	`  Vertex m = Q.removeMin()`
17	`  T.add(m)`
18	`  foreach (Vertex v : neighbors of m not in T):`
19	`  if _____ < d[v]:`
20	`  d[v] = _____`
21	`  p[v] = m`
22	
23	`  return T`

Dijkstra: What if we have a negative-weight cycle?



Dijkstra: What if we have a minimum-weight edge, without having a negative-weight cycle?



Dijkstra makes an assumption:

Dijkstra: What is the running time?

CS 225 – Things To Be Doing:
1. Final Exam runs Thursday, May 3 – Thursday, May 10
2. MP7 is released; EC due tonight, Monday, April 23th
3. Final lab, **lab_ml**, released Wednesday
4. This week is the last week of POTDs *(last POTD is Friday!)*

GPS

CS 225

#39: Dijkstra's Algorithm (for reals!)

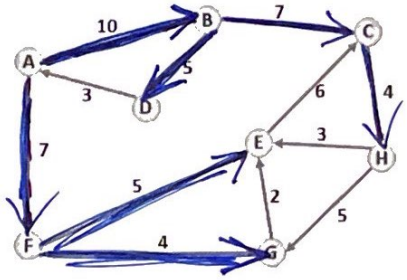
April 25, 2018 · *Wade Fagen-Ulmschneider*

Shortest Path Home:

Dijkstra's Algorithm (Single Source Shortest Path)

priority queue.

A	~~∞~~	
B	~~∞~~	10
C	∞	
D	∞	
E	~~∞~~	7+5=12
F	~~∞~~	7
G	~~∞~~	7+4=11
H	∞	



Dijkstra's Algorithm Overview:

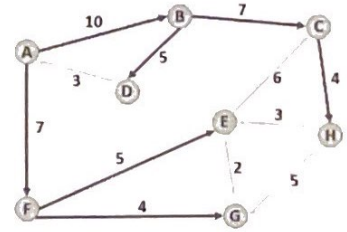
- The overall logic is the same as Prim's Algorithm
- We will modify the code in only two places – both involving the update to the distance metric.
- The result is a directed acyclic graph or DAG

Pseudocode for Dijkstra's SSSP Algorithm

```
1  DijkstraSSSP(G, s):
2    Input: G, Graph;
3           s, vertex in G, starting vertex of algorithm
4    Output: T, DAG w/ shortest paths (and distances) to s
5
6    foreach (Vertex v : G):
7      d[v] = +inf
8      p[v] = NULL
9    d[s] = 0
10
11   PriorityQueue Q   // min distance, defined by d[v]
12   Q.buildHeap(G.vertices())
13   Graph T           // "labeled set"
14
15   repeat n times:
16     Vertex m = Q.removeMin()
17     T.add(m)
18     foreach (Vertex v : neighbors of m not in T):
19       if cost(v,m)+d[m]  < d[v]:
20         d[v] = cost(v,m) + d[m]
21         p[v] = m
22
23   return T
```

Backtracking in Dijkstra

Dijkstra's Algorithm gives us the shortest path from a single source to every connected vertex:



The data structure maintained by Dijkstra's Algorithm will have the following state after running Dijkstra's Algorithm:

	A	B	C	D	E	F	G	H
p	NULL	A	B	B	F	A	F	C
d	0	10	17	15	12	7	11	21

Q: What is the shortest path from A to H?

dist (A, H) = 21 dist (A, E) = 12.

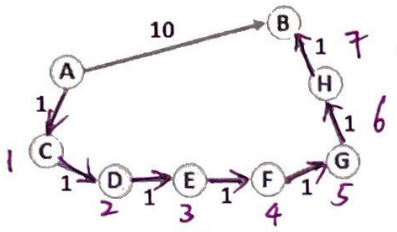
Q: What is the shortest path from A to E?

path (A, H) = A → B → C → H.

path (A, E) = A → F → E.

Examples: How is a single heavy-weight path vs. many light-weight paths handled?

Ex 1:

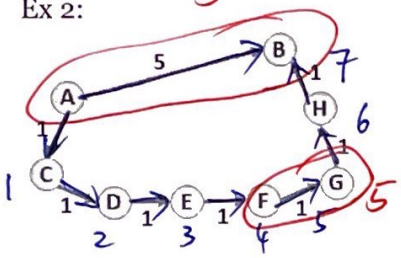


B: ~~P→A, d→10~~

P→H, d→7.

7 B chooses predecessor to be H.

Ex 2:

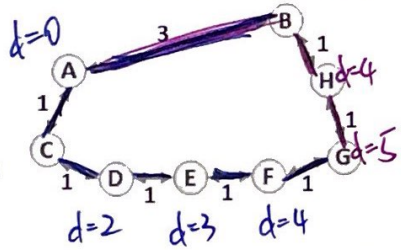


B: P→A, d→5.

7 > 5,

B still chooses A instead of H

What about undirected graphs?

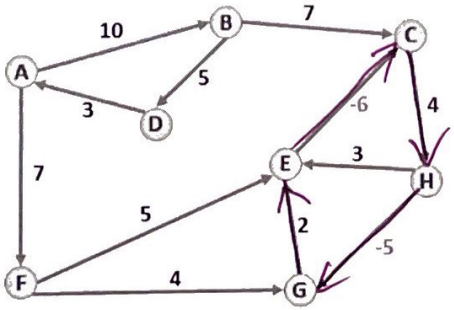


A→G:

A→B→H→G

A→C→D→E→F→G.

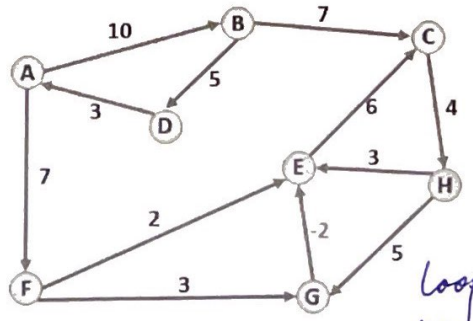
Dijkstra: What if we have a negative-weight cycle?



There is no defined shortest path

Dijkstra: What if we have a minimum-weight edge, without having a negative-weight cycle?

path (A, E) = ?



A→F→E (12)

A→F→(E→C→H→G)*→E

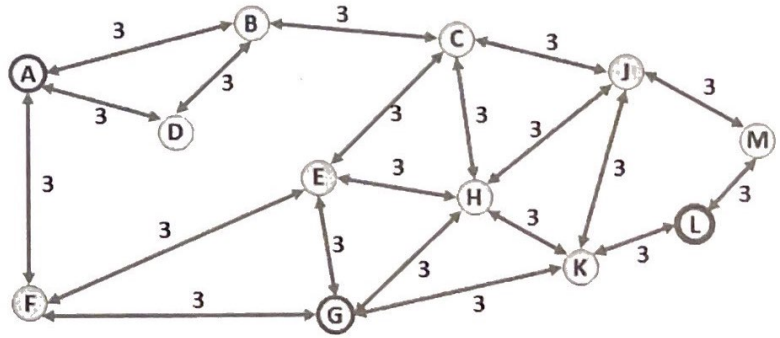
→ = -∞

Loop over the negative-weight cycle repeatedly.

Dijkstra makes an assumption:

Dijkstra Algorithm: What is the running time?

Challenge: Landmark Path Problem



CS 225 – Things To Be Doing:
1. Final Exam runs Thursday, May 3 – Thursday, May 10
2. MP7 is released; MP7 deadline Monday, ~~May 30~~ April 30
3. Final lab, **lab_ml**, released today; due Sunday, ~~May 29~~ April 29.
4. This week is the last week of POTDs *(last POTD is Friday!)*

CS 225

#40: Graph Puzzles + End of Semester

April 27, 2018 · *Wade Fagen-Ulmschneider*

Dijkstra's Algorithm Overview:

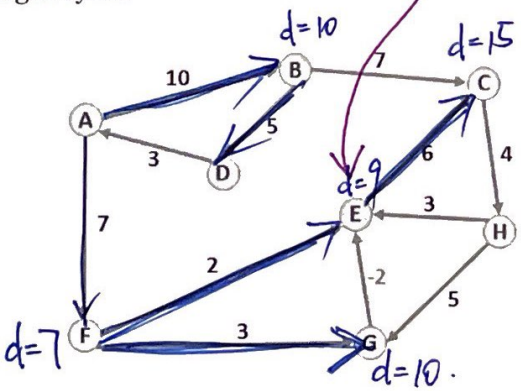
- The overall logic is the same as Prim's Algorithm
- We will modify the code in only two places – both involving the update to the distance metric.
- The result is a directed acyclic graph or DAG containing the shortest path to every vertex from a single starting point.

Pseudocode for Dijkstra's SSSP Algorithm

```
1  DijkstraSSSP(G, s):
6    foreach (Vertex v : G):
7      d[v] = +inf
8      p[v] = NULL
9    d[s] = 0
10
11   PriorityQueue Q    // min distance, defined by d[v]
12   Q.buildHeap(G.vertices())
13   Graph T            // "labeled set"
14
15   repeat n times:
16     Vertex m = Q.removeMin()
17     T.add(m)
18     foreach (Vertex v : neighbors of m not in T):
19       if cost(m,v) + d[m] < d[v]:
20         d[v] = cost(m,v) + d[m]
21         p[v] = m
22
23   return T
```

could not find the true shortest path ⇐ E is already marked. does not update E.

Dijkstra: What if we have a minimum-weight edge, without having a negative-weight cycle?



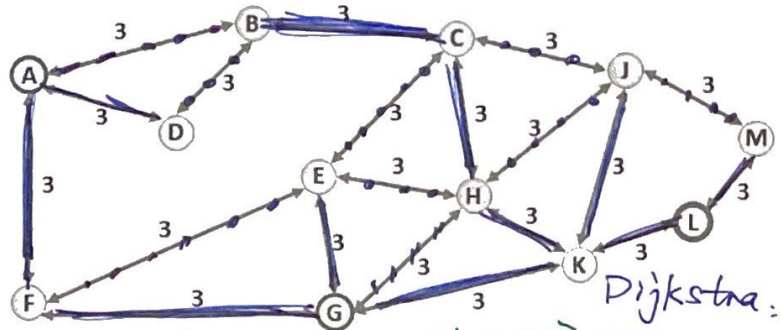
Dijkstra's Algorithm optimality assumption:

non-negative edge weights.

Dijkstra's Algorithm running time:

start with Prim's algorithm. running time depends on what data structure used for graph.
heap Vs. unsorted array Vs. ~~Fabonacci heap~~
adjacency list Vs adjacency matrix.

Challenge #1: Landmark Path Problem



because edge weights are the same

Dijkstra: $n \log n + m$

BFS: $O(n) + O(m)$.

shortest path from A to G: A→F→G

6.

shortest path from A to G passing L.

A→F→G→K→L + L→K→G.

what is the fastest algorithm to find this path?

what are the specific call(s) to this algorithm.

BFS(Graph, L)

programmer Vs. computer scientist

End of Semester Logistics

CS 225 Final Exam

- The final exam begins on Thursday, May 3rd
- The final exam is a 3 hour CBTF exam, is a cumulative exam, and has the format of a combined theory + programming exam
- The last office hours is Wednesday, May 2nd
- *We'll use lecture on Wednesday, May 2nd as a final exam review!*

"Pre-Final" Grade Dump

- I believe there's only a few remaining issues left with grading; I'll be starting to wrap these up myself over the weekend:
  - +EC from creative components
  - Working on recovering repos that were force deleted
- As soon as possible after MP7's deadline, we'll provide a "Pre-Final" grade in Compass that incorporates everything except the final exam into your CS 225 grade.

End of Semester Grade Review

- Excel sheet will be provided once final grades are posted.
- Must submit an Excel sheet for this review.

You're Awesome -- +1 To Your Skills!

- If you earn an A- or better in CS 225, you're awesome! I'll endorse your "Data Structures" skill on LinkedIn if you:
  - Connect with me on LinkedIn
  - Message me a link/images of your creative components on your MPs and let me know you were in CS 225 this semester. I'll endorse your data structures skill and CS225-related skills!

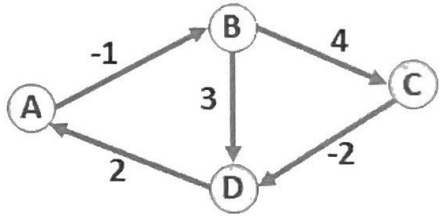
Floyd-Warshall Algorithm

Floyd-Warshall's Algorithm is an alternative to Dijkstra in the presence of negative-weight edges (but <u>not</u> negative weight cycles).

Pseudocode for Floyd-Warshall's Algorithm

```
1  FloydWarshall(G):
2    Input: G, Graph;
3    Output: d, an adjacency matrix of distances between
4  all
5             vertex pairs
6
7    Let d be a adj. matrix initialized to +inf
8    foreach (Vertex v : G):
9      d[v][v] = 0
10   foreach (Edge (u, v) : G):
11     d[u][v] = cost(u, v)
12
13   foreach (Vertex u : G):
14     foreach (Vertex v : G):
15       foreach (Vertex w : G):
16         if d[u, v] > d[u, w] + d[w, v]:
17           d[u, v] = d[u, w] + d[w, v]
18
     return d
```

Running Floyd-Warshall:



	A	B	C	D
A				
B				
C				
D				

CS 225 – Things To Be Doing:

1. Final Exam runs Thursday, May 3 – Thursday, May 10
2. MP7 deadline Monday, April 30
3. Final lab, **lab_ml** due Sunday, April 29
4. Final POTD is right now! ☹

CS 225

#41: Floyd-Warshall's Algorithm

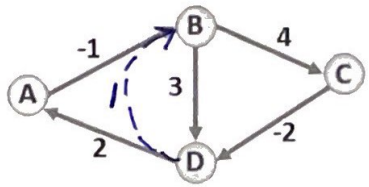
April 30, 2018 · *Wade Fagen-Ulmschneider*

Floyd-Warshall Algorithm

Floyd-Warshall's Algorithm is an alternative to Dijkstra in the presence of negative-weight edges (but not negative weight cycles).

Algorithm Design:

- **Goal:** Find the shortest path from vertex **u** to **v**.
- **Setup:** Create an n×n matrix that maintains the best known path between every pair of vertices:
  - Initialize (u, u) to 0.
  - Initialize all edges present on the graph to their edge weight.
  - Initialize all other edges to +infinity.



shortest known path between 2 vertices

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	~~∞~~ 1	∞	0

- For every vertex **k**, consider which of the following are shorter:
  - path(u, v) - or -
  - path(u, **k**) + path(**k**, v)

k=A. D→B +∞

D→A→B +1

(DP). after our first k loop, all paths are optimal with the addition of A.

Big Idea: Dynamic Programming

- Store intermediate results to improve build towards an optimal solution.
- Example application of memoization.

Running Time:

k=B D→C

D→B→C

D→A→B.

run through entire way to get the final results. $O(n^3)$.

dynamic programming.

Pseudocode for Floyd-Warshall's Algorithm

```
1   FloydWarshall(G):
2     Input: G, Graph;
3     Output: d, an adjacency matrix of distances between
4   all
5             vertex pairs
6
7     Let d be an adj. matrix (2d array) initialized to +inf
8     foreach (Vertex v : G):
9       d[v][v] = 0
10    foreach (Edge (u, v) : G):
11      d[u][v] = cost(u, v)
12
13    foreach (Vertex u : G):
14      foreach (Vertex v : G):
15        foreach (Vertex w : G):
16          if d[u, v] > d[u, w] + d[w, v]:
17            d[u, v] = d[u, w] + d[w, v]
18
      return d
```

n^3

n^3 finds all pairs shortest paths.

Overview of Graphs:

Implementations
- Edge List
- Adjacency Matrix
- Adjacency List

Traversals
- Breadth First
- Depth First

Minimum Spanning Tree
- Kruskal's Algorithm
- Prim's Algorithm

Shortest Path
- Dijkstra's Algorithm
- Floyd-Warshall's Algorithm

...and this is just the beginning. The journey continues to CS 374!

CS 225 – Things To Be Doing:

1. MP7 due tonight (April 30); *standard grace period applies.*
2. Final Exam starts Thursday, May 3