Announcements:

HW1 available tonight, due Jan 20, 11:59p.

Today: 1. intro to algorithm analysis – code! 2. review of asymptotics

Algorithm Analysis – example 1:

```
int indexoffirstq (vector<int>    arr, int q){
    for (int i = 0; i < arr.size(); i++)
        if (arr[i] == q)
            return i;
    return -1;
}
```

- vector – standard template library dynamic array implementation (size can change)
- c++ by default passes by value (a local copy is made when called) of parameter

What does it do?

0	1	2	3	4	5	6	7
32	11	73	21	29	86	81	92

Algorithmic Analysis – running time

```
int ______ (vector<int> & arr, int q){
    for (int i = 0; i < arr.size(); i++)
        if (arr[i] == q)
            return i;
    return -1;
}
```

Comparison may take a long time, eg. comparing pictures

How long does it take? It depends

What does it depend on?

1. size of array
2. clock speed of hardware
3. Location of q in the array

What should we count? steps, lines of code, operations.

Algorithmic Analysis Discussion

```
int ______ (vector<int> & arr, int q){
    for (int i = 0; i < arr.size(); i++)
        if (arr[i] == q)
            return i;
    return -1;
}
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	11	73	21	29	86	81	92	57	61	64	15	79	44	7	45

What's the *best* case running time? q is at arr[0]

What's the *worst* case running time? q is not in the array
longest possible running time

What's the *average* case running time?
requires a probability distribution over the inputs.

Algorithmic Analysis Discussion

```
int ______ (vector<int> & arr, int q){
    for (int i = 0; i < arr.size(); i++)
        if (arr[i] == q)
            return i;
    return -1;
}
```

How many lines are executed in the worst case?

$T(n) =$ 2n+1 (let n=arr.size())

Discussion:
concrete discussion will always lead to inconsequential disagreement

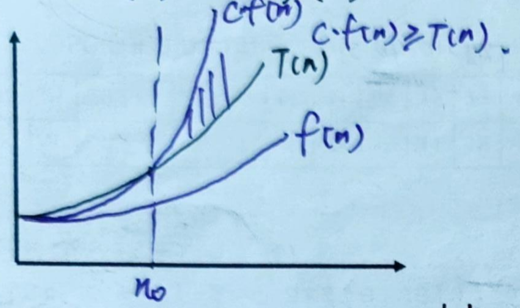
$T(n) = cn + d$ where c, d are constants

$T(n) \in O(n)$

Aside on Asymptotics

running time · reference function

Defn: $T(n) = O(f(n))$ if there are positive constants c and n_0 such that $T(n) \le c \cdot f(n)$ for all $n \ge n_0$.



We typically use f(n) to describe. the asymptotic upper bound on the worst case running time of an algorithm

Asymptotic definitions

- $T(n) \in O(f(n))$ if there are positive constants c and n_0 such that $T(n) \le cf(n)$ for all $n \ge n_0$.
- $T(n) \in \Omega(f(n))$ if there are positive constants c and n_0 such that $T(n) \ge cf(n)$ for all $n \ge n_0$ asymptotic lower bound
- $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$

not seen in the course
- $T(n) \in o(f(n))$ if for **any** positive constant c, there exists n_0 such that $T(n) < cf(n)$ for all $n \ge n_0$.
- $T(n) \in \omega(f(n))$ if for **any** positive constant c, there exists n_0 such that $T(n) > cf(n)$ for all $n \ge n_0$.

The reminder page:

Typical growth rates in order (increasing order of complexity)

- Constant: $O(1)$
- Logarithmic: $O(\log n)$
- Poly-log: $O\left((\log n)^k\right)$
- Linear: $O(n)$
- Log-linear: $O(n \log n)$
- Superlinear: $O(n^{1+c})$ (c is a constant > 0)
- Quadratic: $O(n^2)$
- Cubic: $O(n^3)$
- Polynomial: $O(n^k)$ (k is a constant) "tractable"
- Exponential: $O(c^n)$ (c is a constant > 0) "intractable"

if $f(n) \le O(n \log n)$
then $f(n) \le O(n^{1+c})$

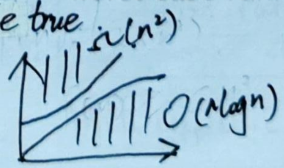
A: $T(n) = \Omega(n^2)$
B: $T(n) = O(n \log n)$
since $A \wedge B = F$.
any conclusion will be true

Algorithmic Analysis Discussion

```
int ______ (vector<int> & arr, int q){
    for (int i = 0; i < arr.size(); i++)
        if (arr[i] == q)
            return i;
    return -1;
}
```

T(n) = __________

Choose which definition we should use for describing worst case running time? $(o, O, \omega, \Omega, \theta, \Theta)$



$f(n) = \Theta(n^2)$
$= \Theta(n\log n$

Announcements:

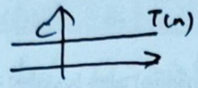
HW1 available, due Jan 20, 11:59p.

Warm-up:

```
int returnfirst (vector<int> & arr, int q){
    for (int i = 0; i < arr.size(); i++)
        if (i == 0)
            return arr[i];
    return -1;
}
```

Good name?

Running time:



O(n) and this T(n) is not Ω(n)

T(n)=c so T(n)=Θ(1)

by definition of O: $T(n) \leq c \cdot 1$

Ω: $T(n) \geq c \cdot 1$.

$\Rightarrow T(n)=\Theta(1)$

by Θ: $T(n)=\Theta(1)$

Runtime analysis guidelines:

always worst case

- Single operations, constant time
- Consecutive operations additive
- Conditionals: constant for Boolean eval + branch time
  and, or, not. worst branch (longest time)
- Loops: sum of loop body times
  Cannot assume that the body of for loop always takes the same amount of time
- Function call: time for function

Algorithmic Analysis Discussion

```
int _firstLoc_ (vector<int> & arr, int q){
    for (int i = 0; i < arr.size(); i++)
        if (arr[i] == q)
            return i;
    return -1;
}
```

T(n) = Cn+d.

Choose which definition we should use for describing worst case running time? ($o, O, \omega, \Omega, \theta, \boxed{\Theta}$)

since $T(n) \leq cn+d$. $T(n) \geq cn+d$.

$T(n) = \Theta(n)$

Some puzzles:

Typical growth rates in order

Tractable

- Constant: $O(1)$
- Logarithmic: $O(\log n)$
- Poly-log: $O\left((\log n)^k\right)$

$O(\sqrt{n})$

- Linear: $O(n)$
- Log-linear: $O(n \log n)$
- Superlinear: $O(n^{1+c})$ (c is a constant > 0)
- Quadratic: $O(n^2)$
- Cubic: $O(n^3)$
- Polynomial: $O(n^k)$ (k is a constant)
- Exponential: $O(c^n)$ (c is a constant > 0) "intractable"

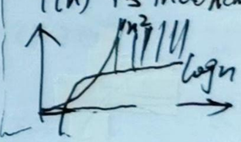
if we take $k>0$ $k \to \infty$.

n^c $c \to 0$.

we still have $n^c > (\log n)^k$

if $T(n)=O(n^2)$ and $T(n)=\Omega(\log n)$

T(n) is inconclusive



Announcements:

HW1 available, due Jan 20, 11:59p.

Warm-up:

```
int findmin (vector<string> & candy, int a){
   int retloc = a;
   string voi = candy[a];
   for (int i = a+1; i < candy.size(); i++)
      if (candy[i]<voi){
            retloc = i;
            voi = candy[i];
      }
   return retloc;
}
```

keep track of the same thing

changed simultaneously

n-a-1 iterations

n-1 iterations in worst case

At iteration i, { retloc holds voi's location; voi holds min in candy[a,i-1]

Good name?

Does it work?

Running time:

$T(n) = d + c(n-1) = \Theta(n)$

let $c_1 = \min(d, c)$

$c_2 = \max\{d, c\}$.

$T(n) \leq (c_2+1)n$ for $n \geq c_2$

$n \geq c_2 \geq d-c$.

$T(n) \geq (c_1-1)n$ for $n \geq c_1$.

Correctness continued:

```
int findmin (vector<string> & candy, int a){
   int retloc = a;
   string voi = candy[a];
   for (int i = a+1; i < candy.size(); i++)
      if (candy[i] < voi){
         retloc = i;
         voi = candy[i];
      }
   return retloc;
}
```

Correctness? (argued formally via induction)

"loop invariant"

Thm: for any i in {a+1,...n-1}, findMin(candy, a) stopped at i gives index of min value in candy[a..i-1]

iterative variable is $i \in \{a+1, \dots n-1\}$

Base case: $i = a+1$ retloc = a voi = candy[a], represent the min of candy[a..a] is true

IH: retloc and voi holds min value in candy[a,... i-1]

start iteration i, if candy[i] ≥ voi, we do nothing, voi, retloc represent min in candy[a,i]

if candy[i] < voi, then voi and retloc are updated. it is still true that retloc and voi represent the minimum in candy[a,..., i]

Proving a (given) loop invariant--

A Induction variable: 5

B Base case: 4

C Induction hypothesis: 1

D Inductive step: 2

E Termination: 3

1. Assume the invariant holds just before beginning some (unspecied) iteration.
2. Prove the invariant holds at the end of that iteration for the next iteration.
3. Make sure the invariant implies correctness when the loop ends.
4. Prove the invariant true before the loop starts.
5. Number of times through the loop.

Proving correctness is only one benefit of loop invariants, they are also a natural way to think about your program!

Another Example

```
void __________ (vector<string> & candy){
   for (int i = 0; i < candy.size(); i++){
      int min = findMin(candy,i);
      string temp = candy[i];
      candy[i] = candy[min];
      candy[min] = temp;}
}
```

Functionality?

Running Time?

Correctness?

1) iterative variable:

2) loop invariant:

Announcements:

HW1 available, due Jan 20, 11:59p.

Warm-up:

```
void selectionSort (vector<string> & candy){
   for (int i = 0; i < candy.size(); i++){
      int min = findMin(candy,i);     ← select min
      string temp = candy[i];   }
      candy[i] = candy[min];    } swap
      candy[min] = temp;}
}
```

running time $\Theta(n-i)$

Good name? Does it work?

Running time:

$$T(n) = \underset{\text{constant}}{1} + \underset{\text{for loop}}{\sum_{i=0}^{n-1}} (\underset{\text{findmin}}{n-i} + \underset{\text{constant}}{1}) = 1 + \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1}(n-i)$$

$$= 1 + n + \frac{n(n+1)}{2} = \Theta(n^2)$$

Another Example

```
void selectionSort (vector<string> & candy){
   for (int i = 0; i < candy.size(); i++){
      int min = findMin(candy,i);
      string temp = candy[i];
      candy[i] = candy[min];
      candy[min] = temp;}
}
```

Correctness?

1) iterative variable:
   $i \in \{0, 1, 2, \cdots, n\}$

2) loop invariant: (start condition at iteration i)
   A: candy [0,..i-1] is sorted in non-decreasing order
   B: $x \in \{0,..i-1\}$, $y \in \{i,.. n-1\}$, then candy[x] ≤ candy[y]
   for all i at the start of iteration i

Selection Sort: (correctness continued)

```
void selectionSort (vector<string> & candy){
   for (int i = 0; i < candy.size(); i++){
      int min = findMin(candy,i);
      swap(candy[i], candy[min]);}
}
```

3) Base case (LI true at start of first iteration): loop invariant
   i=0
   A. candy [0,…,-1] is sorted ✓ (empty arrays are sorted)
   B $x \in \{0,..-1\}$ $y \in \{0,…,n-1\}$ ⇒ candy[x] ≤ candy[y] ✓

4) IH (assume LI true at start of ith iteration):

5) Inductive step (show true at start of step i+1): (code restores loop invariants)
   1. candy [min] ≤ candy [i,.. n-1] (correctness of findmin)
   2. candy [min] ≥ candy [0,…: i-1] by B.
   3. so swap moves candy [min] to candy[i] and vice versa.
   Now candy[0,.. i-1, i] is sorted by A + 2, candy[i+1, … n-1] are all larger elements by B.

6) Termination (last value of iterator): n.
   A: candy[0,…, n-1] is sorted, B not needed.

Weird mystery function

```
void ___________ (vector<string> & candy, int loc){
// assumes candy[0:loc-1] is sorted, loc valid
    string temp = candy[loc];
    int j = loc;
    while (j > 0 && candy[j-1] > temp) {
        candy[j] = candy[j-1];
        j--;  }
    candy[j] = temp;
}
```


Functionality?

Running Time?

Correctness?

1) Iterative variable:

2) Loop Invariant:

Announcements:

HW1 due today 11:59p. PA1 due 02/03, 11:59p. Quizzes

Warm-up: Weird Mystery Function

```
void slide (vector<string> & candy, int loc){
// assumes candy[0:loc-1] is sorted, loc valid
    string temp = candy[loc];        temp=C.
    int j = loc;                     j=5
    while (j > 0 && candy[j-1] > temp) {
        candy[j] = candy[j-1];
        j--;  }
    candy[j] = temp;
}
```

loc

candy

0	1	2	3	4	5	6	7
A	D	E	H	L	C	J	R

largest loc for this candy is 5

A C D E H L | J R.

Good name? Does it work?

Running time: $\Theta(n)$ if standalone

$\Theta(loc)$ since loc determined by insertionsort.

assume candy[0:loc-1] is sorted.

Sort values candy[0...loc].

You write Insertion Sort…

```
void insertionSort (vector<string> & candy){
    for( int i=1; i< candy.size(); i++){
        slide(candy, i);
    }
}
```

0	1	2	3
M	A	T	H

Functionality? insertion sort.

Running Time?

$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \Theta(n^2)$

Correctness?

1) Iterative variable: $i \in \{1, \dots, n\}$ ← not happen

2) Loop invariant:

candy[0... i-1] is sorted.

Insertion Sort

```
1  vector<string> insertionSort (vector<string> & candy){
2      for (int i = 1; i < candy.size(); i++) {
3          slide(candy, i);
4  return candy; }
```

3) Base case: i=1

candy[0, ..., i-1] = candy[0...0) is sorted.

4) IH: at start of iteration i, candy[0, ..., i-1] is sorted.

5) Inductive step:

at start of iteration i, candy[0, ..., i-1] is sorted.

by correctness of slide, after line 3, candy[0, ..., i] is sorted.

LI restored for iteration i+1

6) Termination: i=n.

LI gives candy[0, ..., n-1] is sorted.

Linear Sorts, recap

We have learned and analyzed *selection sort* and *insertion sort*. Which is better?

- Asymptotically? the same $\Theta(n^2)$.
- Empirically? (with data, practically) selection
- What if list is already sorted? ~~sele~~ insertion
- What if list is almost sorted? selection
- What if list is in reverse order? selection

https://www.toptal.com/developers/sorting-algorithms

Something NEW!!

Make at least 3 observations about this code:

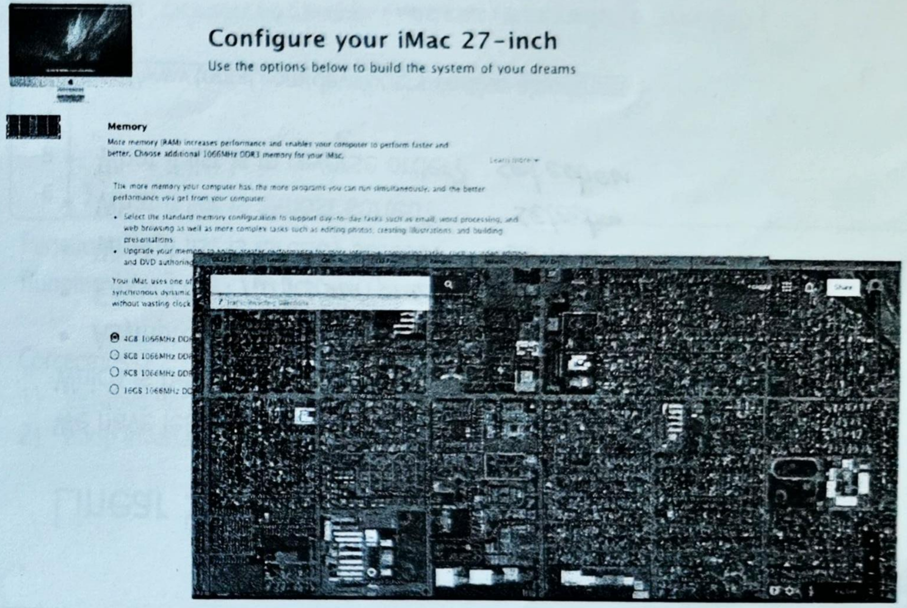
```
1  template <class LIT>
2  struct Node{
3     LIT data;
4     Node * next;
5     Node(LIT nData, Node * next=NULL): data(nData) {}
6  };
```

struct is the same as class

syntax for default

- recursive struct
- line 1 parameterizes type info
- struct is class with public default access
- line 5 is a constructor

Switching gears… for line 4:



Announcements:

PA1 due 02/03, 11:59p. Quizzes ongoing.

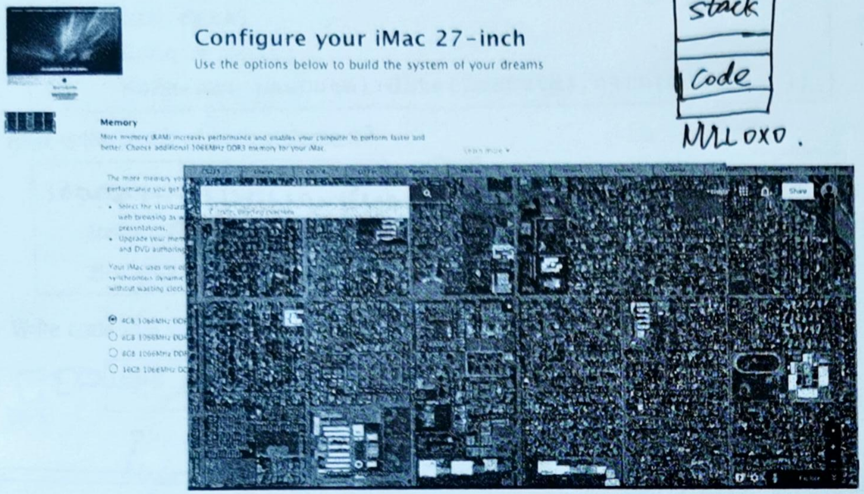
Memory

Heap

stack

code

NULL 0x0.



Variables and Pointers:

Backtrack to variables:

int x; ← before assigning, val is garbage
x=15;
Node<int> tyler; //const invoked
int y;

Special variable type:
(pointer == memory address)
int *p;

$\boxed{\text{p=x; is a compiler error (type mismatch)}}$

p=&x; (& is memory address operator, gives the memory address of x)

Stack

every variable has

loc	name	val	type
a16	p	mem addr	int*
a20	y		int
a36	tyler	int Node ptr	Node<int>
a40	x	15	int

Pointers and dynamic memory:

int x;
int * p;
p= &x;

42 p x

cout << $\boxed{\text{*p}}$ << endl; 15

* unary operator: follows pointer and returns the target.

p = new int;
*p=42;

Stack

loc	name	val	type
a36	p	b12 ~~a40~~	int*
a40	x	15	int

Heap

loc	name	val	type
b12		42	int

Fun and games with pointers: (warm-up)

```
int * p, q;
```

What type is q? int

```
int *p;
int x;
p = &x;
*p = 6;
```

x=6

p x

```
cout << x;
```

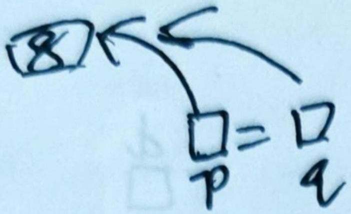
What is output? 6

```
cout << p;
```

What is output? address of x.

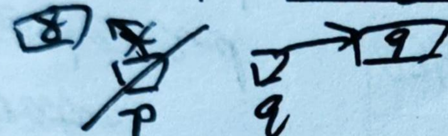
Write a statement whose output is the value of x, using variable p: cout<<*p;

```
int *p, *q;
```




```
p = new int;
q = p;
*q = 8;
cout << *p;
```

What is output? 8




```
q = new int;
*q = 9;
p = NULL;
```

Do you like this? X memory leak

```
delete q;
```

give the memory location at q back to the system.

```
q = NULL;
```

Do you like this? ✓

pull back the point

if *q=30, runtime error, seg fault

Memory leak: occurs when all pointers to same heap memory are removed.

Deleting a null pointer: nothing happens

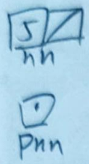
Dereferencing a null pointer: seg fault, runtime error.

Announcements: PA1 due 02/03, 11:59p. Quizzes ongoing.

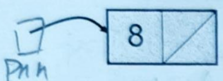
```
template<class LIT>
struct Node{
   LIT data;
   Node * next;
   Node(LIT newData):data(newData),next(NULL){} };
```

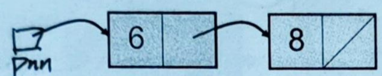
What is the result of each statement?

```
int main(){
   Node<int> nn(5);
   Node<int> * pnn;   }
```



Write code that would result in each of these memory configurations (in order):



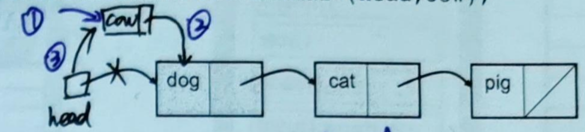



```
pnn = new Node<int>(8);
Node<int>* n1 = new Node<int>(6);
  Node<int>* tmp = pnn;
pnn = n1;
pnn -> next = tmp;
```

```
or Node* t = new Node(6);
        t->next = pnn;
        pnn = t;
```

Example 1:

`insertAtFront<farmAnimal>(head,cow);`



pass by reference

```
void insertAtFront(Node *& curr, LIT e) {
    Node<LIT> * t = new Node<LIT>(e);   ①
    t->next = curr;   ②
    curr = t;   ③
}
```

Running time?

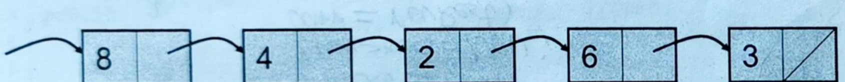
$\Theta(1)$

```
struct Node{
   LIT data;
   Node * next;
   Node(LIT newData):data(newData),next(NULL){}
};
```

Example 2:

8 4 2 6 3

outcome: 3 6 2 4 8




```
void printReverse(Node *   curr) {
   if (curr != NULL) {
      printReverse(curr->next);
      cout << curr -> data;
   }
}
```

T(n): running time on list of len n

Running time?

$T(0) = C$

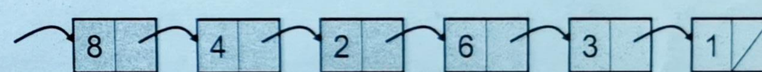
$T(n) = C + T(n-1)$

$= \Theta(n)$

```
struct Node{
   LIT data;
   Node * next;
   Node(LIT newData):data(newData),next(NULL){}
};
```

Example 3:

8 4 2 6 3 1 → 3 2 8

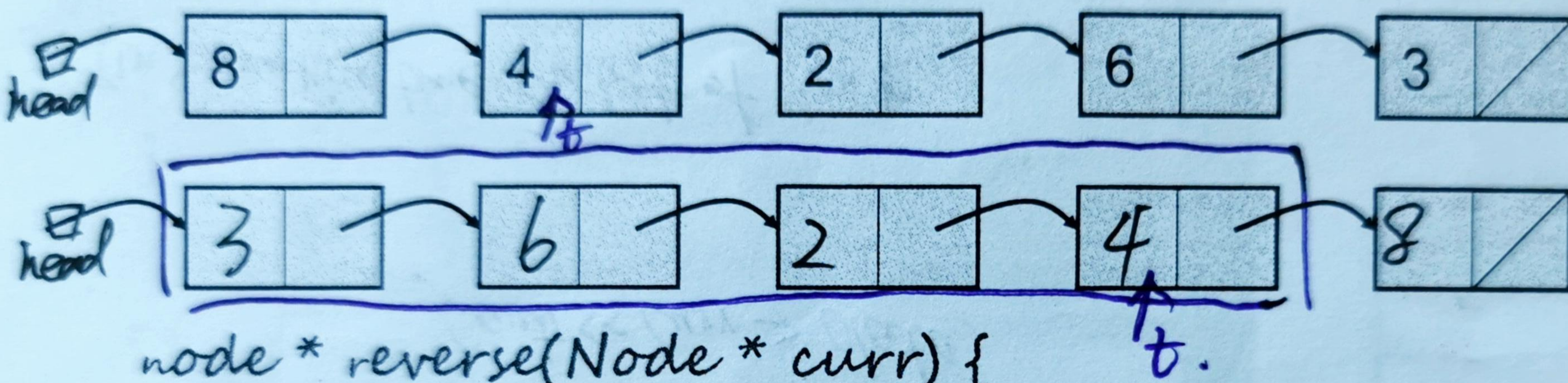



```
void printReverseODDS(Node * curr) {
    // no items in list
    if(curr == NULL){
        return;
    }
    // one item in list
    else if (curr->next==NULL){ cout<< curr->data;}

    // > 1 item in list
    else{ printReverseODDS(curr->next->next);
          cout<< curr->data; }
}
```

Running time? $T(0) = C \quad T(1) = C$

$T(n) = C + T(n-2) = \Theta(n)$

Example 4: head = reverse(head);



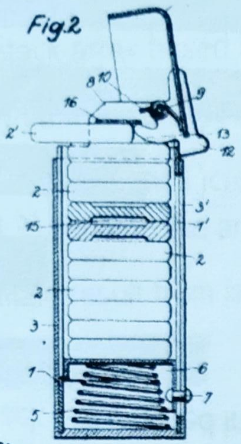

```
node * reverse(Node * curr) {
    if (curr!=NULL && curr->next != NULL) {
zero element                        one element
        Node * temp = curr->next;
        Node * revRest = reverse(curr->next);
           t->next = curr;
           curr->next = NULL;
           curr = revRest;
    }
    return curr;
}
```

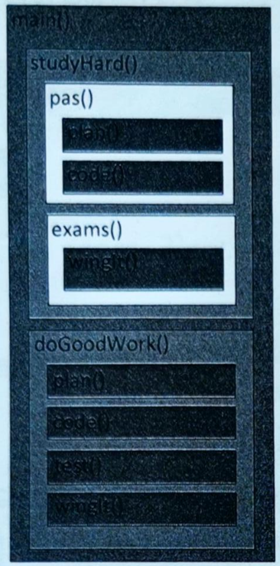
Running time? $\Theta(n)$

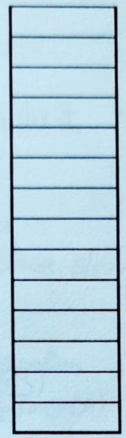
Announcements: PA1, 2/3. Exam1, 2/5.

Stacks

Fig.2







if left then push ;
if right remove and match ;

({ } ( ) [(( )) { (( )) ( ) }] ( ))

4 5 + 7 2 - * 3 - 6 /

Abstract Data Type: *description of the functionality of a structure (interface)*

Stack ADT:

```
template<class LIT>
class Stack{
public:
   Stack(); // ~Stack, cctor, op=?
   bool empty() const;
   void push(const LIT & e); insert.
   LIT pop(); remove, return data.
private:

   ?

};
```

push(3)
push(8)
push(4)
pop() → 4
pop() → 8
push(6)
pop() → 6
push(2)
pop() → 2
pop() → 3.

Push 1, 2, 3 in order. What output is impossible?

3 1 2 is impossible.

Stack linked memory implementation:

```
template<class LIT>
class Stack{
public:
   Stack(); // ~Stack, cctor, op=?
   bool empty() const;
   void push(const LIT & e);
   LIT pop();
private:
    struct Node {
       LIT data;
       Node * next;};
    Node * top;
    int size;
};
```

Running time.

T(pop) = T(push)
= Θ(1)

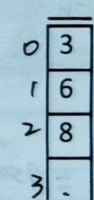

```
template<class LIT>
LIT Stack<LIT>::pop(){
   assert(!empty());
   LIT retval = top->data;
   <Node* tmp=top;
   top=top->next;
   <delete tmp;
   return retval;
   size--;
   tmp=NULL;
}
```

insertAtFront.

```
template<class LIT>
void Stack<LIT>::push(L ... {
Node* newN = new Node(d
newN->next = top
top = newN; size++;
}
```

Stack array based implementation:

```
template<class LIT>
class Stack{
public:
   Stack(); // ~Stack, cctor, op=?
   bool empty() const;
   void push(const LIT & e);
   LIT pop();
private:
    vector<LIT> items;
};
```



top of stack
items[items.size()-1]

```
template<class LIT>
LIT Stack<LIT>::pop(){  assert(!empty());
   LIT retval = items[items.size()-1];
   items.pop_back();
   return retval;
}
```

T(pop) = Θ(1)

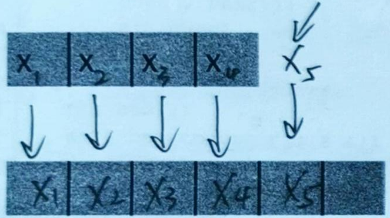
```
template<class LIT>
void Stack<LIT>::push(LIT d){

   items.push_back(d);

}
```

Stack array based implementation: (what if array fills?)

How do vectors work? STL implementation of a dynamic resizing array

What does it mean for an array to be "dynamic"? detects when it needs more space and grabs it

Analysis holds for array based implementations of Lists, Stacks, Queues, Heaps…

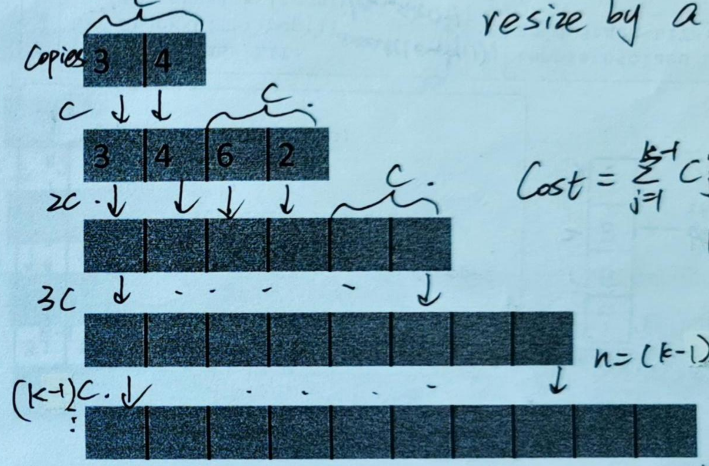


General Idea: upon an insert (push), if the array is full, create a larger space and copy the data into it.

Main question: What's the resizing scheme? We examine 2…

Stack array based implementation: (what if array fills?)

resize by a constant.



$$Cost = \sum_{j=1}^{k-1} Cj = \frac{Ck(k-1)}{2}$$

$$k = \frac{n}{C}$$

$n = (k-1)C$

$n = kC$

How does this scheme do on a sequence of n pushes?

$$Cost = C \cdot \frac{\frac{n}{C}\left(\frac{n}{C}-1\right)}{2} = \Theta(n^2)$$

over n additions $\Rightarrow$ avg of $\Theta(n)$ per addition.

Announcements: PA1, 2/3. Exam1, 2/5.

Array Resizing fin...

…Stack array based implementation:

…What if the array fills?

…General idea: 1)create larger space, 2)copy data into it, and 3) rename the array.

…Main result from last time? analysis over n additions sometimes fast, sometimes slow (amortized analysis)

x	x	x	x

$\Theta(n^2)$ cost over n operation

$\Rightarrow$ an average of $\Theta(\frac{n^2}{n}) = \Theta(n)$ per operation.

Another resizing strategy: (current size) × 2 when array fills

1

2

4

⋮ 2^{k-1}

$$Cost = \sum_{j=0}^{k-1} 2^j$$
$$= 2^k - 1$$
$$= n - 1$$
$$= \Theta(n).$$

- when poping, pop to $\frac{1}{4}n$, then shrink the size to $\frac{1}{2}n$
- if we grow by an arbitrary constant, the base changes $n = C^k$.

$n = 2^k$ | we resize $\log_2 n$ times.

How does this scheme do on a sequence of n pushes? $\Theta(n)$ over n pushes

$\Theta(\frac{n}{n}) = \Theta(1)$ per push.

Stack Summary:

Linked list based implementation of a stack:

Constant time push and pop.

Array based implementation of a stack:

$\Theta(1)$ time pop. if not too empty

$\Theta(1)$ time push if capacity exists,

Cost over O(n) ~~pushes~~ changes is $\Theta(n)$ for an AVERAGE of $\Theta(1)$ per ~~push.~~ change

Why consider an array? Faster in practice. (array is a block of memory)

tracing down the pointer for the linkedlist takes time.

if array takes 1min, the linkedlist takes 10 min

Queues

Queue ADT:

```
enqueue
dequeue
isEmpty
```

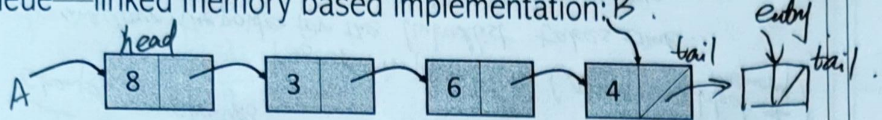
entrance is the back → 4 3 2 1 → exit is the front.

Applications:

- Hold jobs for a printer
- Store packets on network routers
- Make waitlists fair
- Breadth first search

```
enqueue(1)
enqueue(2)
enqueue(3)
dequeue() → 1
dequeue() → 2
enqueue(4)
```

Queue—linked memory based implementation: B .




```
template<class SIT>
class Queue {
public:
// ctors dtor
bool empty() const;
void enqueue(const SIT & e);
SIT dequeue();
private:
    struct queueNode {
        SIT data;
        queueNode * next;
    };
    queueNode * entry;
    queueNode * exit;
    int size;
};
```

Which ptr is "entry", which is "exit"?

B A.

$\boxed{\text{if B is exit, dequeue need to set a pointer with } \Theta(n) \text{ operation}}$

What is running time of enqueue?

$\Theta(1)$

What is running time of dequeue?

$\Theta(1)$

Queue array based implementation:

~~3~~ 8	~~8~~ 4	~~4~~					

enqueue(3);
enqueue(8);
enqueue(4);
dequeue(); → 3
enqueue(7);
dequeue();
dequeue();
enqueue(2);
enqueue(1);
enqueue(3);
enqueue(5);
dequeue();
enqueue(9);

shifting data is bad

```
template<class SIT>
class Queue {
public:
// ctors dtor
bool empty() const;
void enqueue(const SIT & e);
SIT dequeue();
private:

vector<SIT> items;
int entry; } array index
int exit;
};
```

8 ~~3~~ 4	8 ~~3~~	3					

entry entry

~~3~~	8	4	7				

exit . use indicator

Queue array based implementation:

entry exit exit . entry entry

a	y	i	s	~~m~~ h	o	n	d

(entry++)%capacity

…	
enqueue(m);	enqueue(y);
enqueue(o);	enqueue(i);
enqueue(n);	enqueue(s);
…	dequeue();
enqueue(d);	enqueue(h);
enqueue(a);	enqueue(a);

double the array start with the exit.

```
template<class SIT>
class Queue {
public:
// ctors dtor
bool empty() const;
void enqueue(const SIT & e);
SIT dequeue();
pri

    int size;
    vector<SIT> items;

};
```

o	n	d	a	y	i	s	h	a							

exit entry

Circular array

Something new…

Given: an array ______________________

Output: a sorted list

-4	3	5	7	1	2	6	9

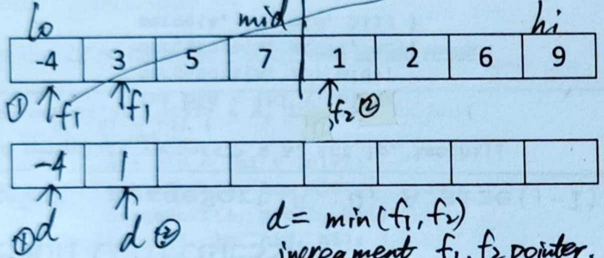
Announcements: PA1, 2/3. Exam1, 2/5.

Something New:

Given: an array that we can divide into 2 sorted parts

Output: a sorted list

merge (Vector A, int lo, int mid, int hi)



$d = \min(f_1, f_2)$

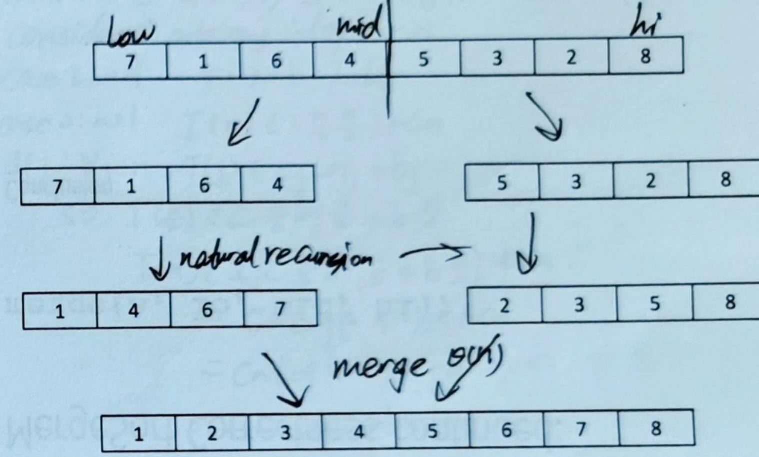
increament f_1, f_2 pointer.

Running Time? n: # of data elements we want to sort

$\Theta(n)$

OK, then sort this array:

merge sort



mergeSort

A "divide and conquer" algorithm. 1) solve smaller subproblems 2) reassemble solutions into the solution to original problem.

1. If the slice of interest of the array has 0 or 1 elements, it's sorted. Stop.
2. Split the array into two approximately equal-sized halves.
3. Sort each half recursively
4. Merge the sorted halves to produce one sorted result.

```
void mergeSort(vector<T> & A, int lo, int hi){
    if (hi > lo) {
        int mid = (hi + lo)/2;
        mergeSort(A, lo, mid);
        mergeSort(A, mid+1, hi);
        merge(A, lo, mid, hi); }
}
```

Base $hi \le lo$

0, 1 element

RT: $T(n)$: running time of merge sort on data of size $n = hi - lo + 1$

$T(0) = T(1) = b$

$T(n) \le \underbrace{2T(\tfrac{n}{2})}_{\text{two mergesort}} + \underbrace{cn}_{\text{merge}}, \quad n > 1$

Finding a closed form (two approaches, there are others):

no recursive term and no sums

1) Expand and generalize: $T(0) = T(1) = b$

$T(n) \le 2T(\frac{n}{2}) + cn \qquad T(\frac{n}{2}) \le 2T(\frac{n}{4}) + \frac{cn}{2}$

$\le 2(2T(\frac{n}{4}) + \frac{cn}{2}) + cn. \qquad T(\frac{n}{4}) \le 2T(\frac{n}{8}) + \frac{cn}{4}$

$= 4T(\frac{n}{4}) + 2cn.$

$\le 4(2T(\frac{n}{8}) + \frac{cn}{4}) + 2cn$

$= 8T(\frac{n}{8}) + 3cn.$

pattern. $= 2^k T(\frac{n}{2^k}) + kcn.$ express the pattern in terms of recursive variable.

force the base case ($\frac{n}{2^k} = 1$)

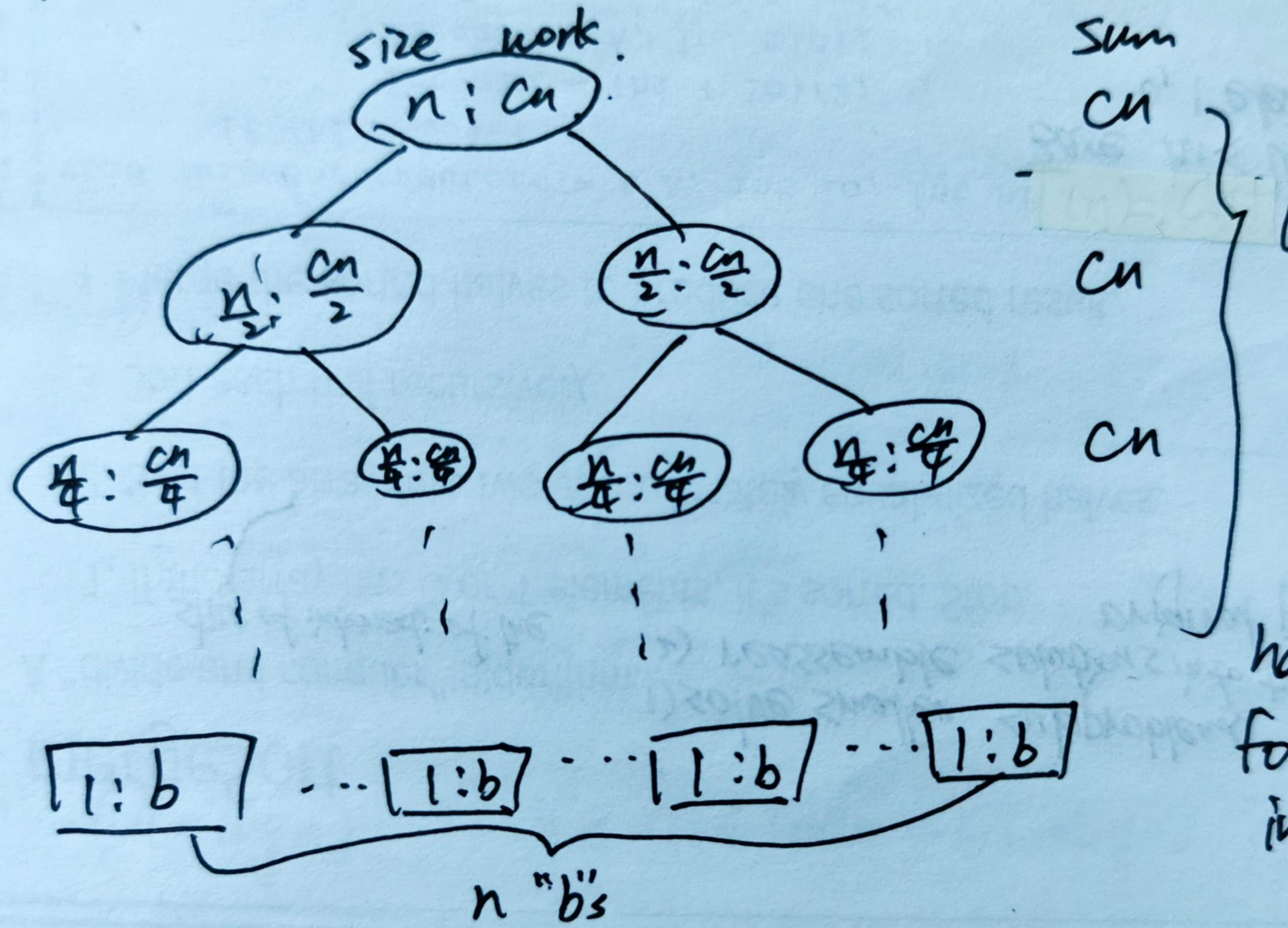
$n = 2^k \cdot k = \log_2 n$

$T(n) \le nb + \log_2 n \cdot c \cdot n$

$T(n) = O(n \log n)$

Finding a closed form (two approaches, there are others):

2) Recursion Tree: $T(n) \leq 2T(\frac{n}{2}) + cn$.



we have $\log_2 n$ levels each of which has cn work for $cn\log_2 n + bn$ in total

Announcements: PA1, today! Exam1, 2/5.

mergeSort:

A "divide and conquer" algorithm.

1. If the array has 0 or 1 elements, it's sorted. Stop.
2. Split the array into two approximately equal-sized halves.
3. Sort each half recursively
4. Merge the sorted halves to produce one sorted result.

```
void mergeSort(vector<T> & A, int lo, int hi){
      if (hi > lo) {
            int mid = (hi + lo)/2;
            mergeSort(A, lo, mid);
            mergeSort(A, mid+1, hi);
            merge(A, lo, mid, hi); }
}
```

RT: $O(n\log n)$ $T(n) \leq 2T(\frac{n}{2}) + cn = O(n\log n)$

Proving the closed form is correct:

$T(n) \leq 2T(\frac{n}{2}) + cn$

$T(0) = T(1) = b$. given

Thm: $\forall n \geq 1, T(n) \leq cn\log n + bn$. $\Rightarrow T(n) = O(n\log n)$

consider an arbitrary integer $n \geq 1$

Case 1: $n = 1$ $T(1) = b$ holds

case 2: $n > 1$ $T(n) \leq 2T(\frac{n}{2}) + cn$

IH: $\forall j < n$, $T(j) \leq cj\log j + bj$

so $T(\frac{n}{2}) \leq c\cdot\frac{n}{2}\log\frac{n}{2} + b\cdot\frac{n}{2}$

$T(n) \leq 2\left[c\cdot\frac{n}{2}\log\frac{n}{2} + b\cdot\frac{n}{2}\right] + cn.$

$= cn\log\frac{n}{2} + bn + cn.$

$= cn\log n - \cancel{cn} + bn + \cancel{cn}$

$= cn\log n + bn$

MergeSort Correctness:

Call MergeSort: `mergeSort(A, 0, A.size()-1);`

```
void mergeSort(vector<T> & A, int lo, int hi){
      if (hi > lo) {
            int mid = (hi + lo)/2;
            mergeSort(A, lo, mid);
            mergeSort(A, mid+1, hi);
            merge(A, lo, mid, hi); }
}
```

Claim: mergeSort correctly sorts A[lo..hi] for any $(hi-lo+1) \in \mathbb{Z}$

Let n = (hi − lo + 1), arb

$n \leq 1$: (Base Case) $hi - lo + 1 \leq 1$, $hi \leq lo$, if $hi < lo$ no element, if $hi == lo$ 1 element. nothing is done.

$n > 1$: IH: Assume mergeSort correctly sorts a slice of size j, for any $j < n$.

`int mid = (hi + lo)/2;` $mid \geq lo$ and $mid < hi$.

`mergeSort(A, lo, mid);` sorts A[lo, … , mid]. because size is smaller (by IH)

`mergeSort(A, mid+1, hi);` sorts A[mid+1, …, hi].

MergeSort Correctness continued:

`merge(A, lo, mid, hi);`

merge demands A[lo… mid] and A[mid+1, hi] sorted and returns A[lo, … hi] sorted.

Conclusion:

mergeSort(A,lo,hi) works for any size.

$\Rightarrow$ mergeSort(A, 0, A.size−1) sorts A.

To contemplate:

does the value of mid matter in the correctness proof? yes, but it only needs to be somewhere $lo \leq mid < hi$.

does the value of mid matter in the analysis of the runtime?

yes.

Where are we in the sorting picture?

	Best Case	Average Case	Worst Case
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge	$\Theta(n)$	$\Theta(n\log n)$	$O(n\log n)$.

no merge needed.

https://www.toptal.com/developers/sorting-algorithms

Complexity of the Sorting Problem:

The *complexity* of a problem is the runtime of the *fastest* algorithm for that problem.

We'll only consider comparison-based algorithms. They can compare two array elements in constant time.

They cannot manipulate array elements in any other way.

For example, they cannot assume that the elements are numbers and perform arithmetic operations (like division) on them.

Insertion, Merge, Quick, ~~Radix~~, Selection

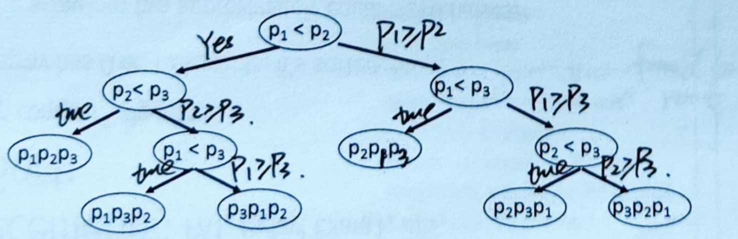
Lower bound on sorting:

An example – different encoding of a sorting procedure

Spose you have an unsorted array of length 3

Need to produce a result for every possible arrangement (p_1 p_2 p_3)

Fundamental operation: Compare 2 elements using <



need n! leaves ~~fo~~ or ends to a sequence of comparison.

Lower bounds on (< based) sorting algorithms (n items):

Consider a decision tree that arises from a comparison based sorting algorithm on *n* items:

- Every permutation must be differentiated in the algorithm
- There are ______ permutations.
- Comparisons induce a *binary tree* whose height determines the running time of the algorithm.
- Observe that a binary tree of height k has at most ____ leaves.

What is the minimum height of a comparison tree for sorting input of size n?

Announcements: Exam1, 2/5.

Lower bound on sorting:

$T(n) \le 2T(\frac{n}{2}) + cn$

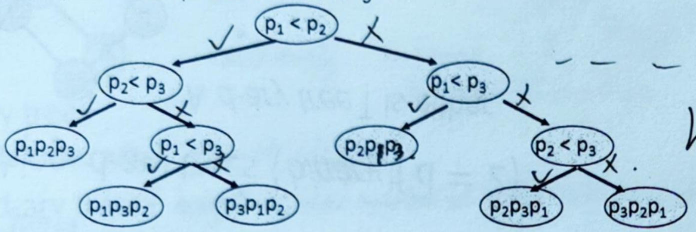
$T(0) = T(1) = b$ $T(n) = \Theta(n \log n)$

An example – different encoding of a sorting procedure

Spose you have an unsorted array of length n

Need to produce a result for every possible arrangement ($p_1\ p_2 \dots p_n$)

Fundamental operation: Compare 2 elements using <



worst case running time of the alogrithm corresponds to the height of tree.

~~Enough terminal nodes to differentiate n! arrangement~~

Lower bounds on (< based) sorting algorithms (n items):

no sorting algorithm can be faster

Consider a decision tree that arises from a comparison based sorting algorithm on *n* items:

- Every permutation must be differentiated in the algorithm
- There are <u>n!</u> permutations.
- Comparisons induce a *binary tree* whose height determines the running time of the algorithm.
- Observe that a binary tree of height h has at most 2^h leaves. (terminal nodes)

h=0 1
h=1 2.
h=2 4.

What is the minimum height of a comparison tree for sorting input of size n? $2^h \ge n!$

$h \ge \log n! = \sum_{k=1}^{n} \log k$

$= \sum_{k=1}^{n/2} \log_2 k + \sum_{k=\frac{n}{2}+1}^{n} \log_2 k.$

$\ge \sum_{k=\frac{n}{2}+1}^{n} \log_2 k.$

$\ge \frac{n}{2} \log_2 (\frac{n}{2}+1) \ge \frac{n}{2} \log_2 (\frac{n}{2}) = \frac{n}{2} \log_2 n - \frac{n}{2}$

$\sum_{k=1}^{n} \log k \le n \log n.$

so $h = \Omega(n \log n)$. is the lowest time any sorting algorith can reach.

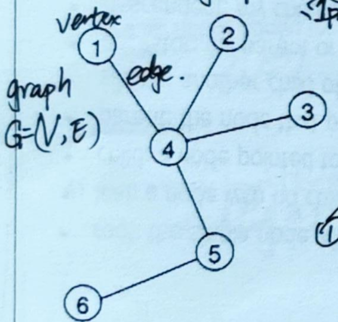
Trees: "… most important nonlinear structure[s] in computer science."

-- Donald Knuth, *Art of Computer Programming Vol 1*

A tree: <u>A graph acyclic and connected.</u>

≤1 path between two nodes   ≥1 path between two nodes

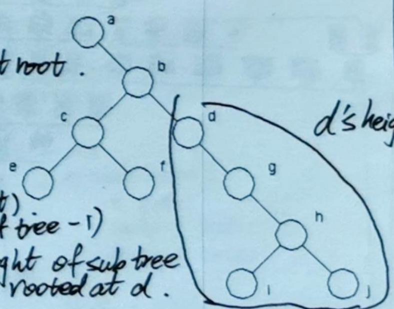
We'll study more specific trees: rooted, directed, ordered. (root is always at the top)



Family Trees
Organization Charts
Classification trees
  What kind of flower is this?
  Is this mushroom poisonous?
File directory structure:
  folders, subfolders in Windows
  directories, subdirectories in UNIX
Non-recursive call graphs

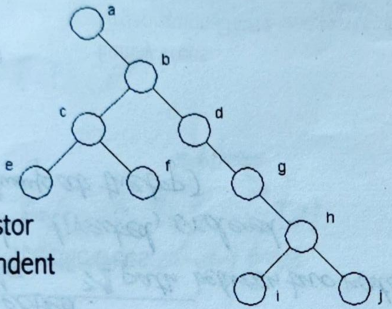
Tree terminology:

- One of the vertices is called the root of the tree. Guess which one it is. a.
- Make an English word containing the names of the vertices that have a parent but no sibling. b, g, h.
- How many parents does each vertex have? 1 except root.
- Which vertex has the fewest children? e,f,i,j
- Which vertex has the most ancestors? descendants? i,j (5 ancestors) a (root) (size of tree - 1)
- What is d's depth? What is d's height? 2 3 (from d to root) (height of sub tree rooted at d.)
- List all the vertices is b's left subtree. c,e,f.
- List all the leaves in the tree. e,f, i, j

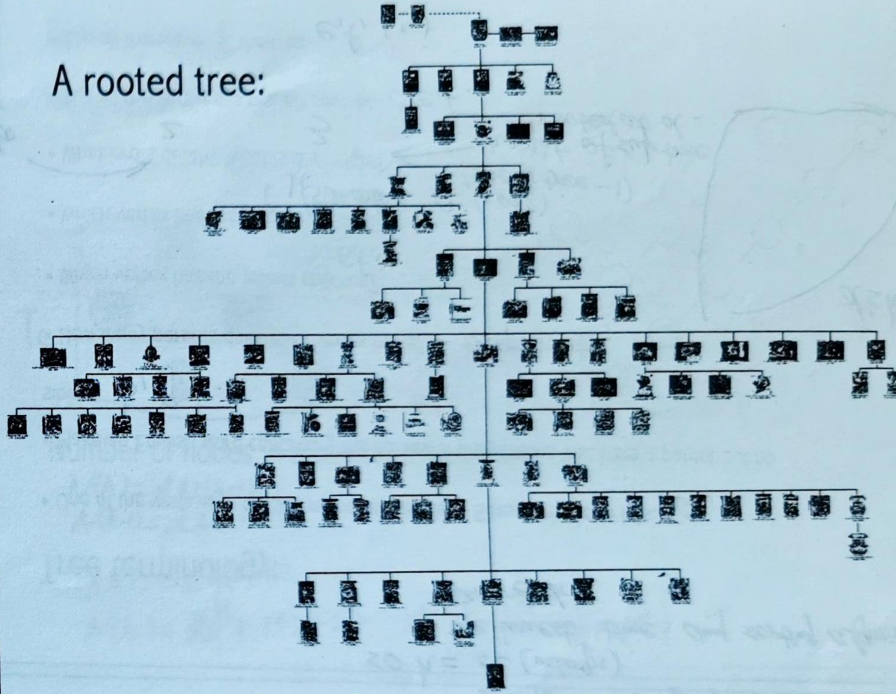


Tree terminology: (for your reference)

- root: the single node with no parent
- leaf: a node with no children
- child: a node pointed to by me
- parent: the node that points to me
- sibling: another child of my parent
- ancestor: my parent or my parent's ancestor
- descendent: my child or my child's descendent
- subtree: a node and its descendents
- depth of node x: number of edges on path from root to x.
- height of node x: number of edges on longest path from x to a leaf.

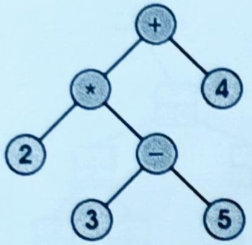


A rooted tree:



Announcements: HW2 due 2/14.

Branching: d-ary trees (*binary* if d = 2)



A *d-ary tree* T is either

- empty $T=\{\}, \emptyset$

OR

- not-empty. $T=\{root, T_1, T_2, \cdots T_d\}$. subtrees
  T_k are d-ary trees

Full d-ary tree: every node has either 0 or d children.

Perfect d-ary tree of height h: max number of nodes in a tree of height h

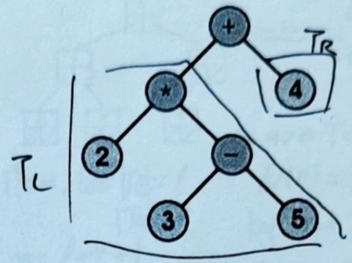
$P(0)=1$

$P(h)=dP(h-1)+1$

Complete d-ary tree: like a perfect tree except deepest level may be missing nodes from the right.

Binary Tree Height

height(r) -- length of longest path from root r to a leaf



Given a binary tree T, write a recursive defn of the height of T, height(T):

- $T=\{\}$ empty $h(T)=-1$
- $T=\{r, T_L, T_R\}$
  $h(T)=\max\{T_L, T_R\}+1$

Number of nodes in a perfect tree of height h, N(h):

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

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

$\vdots$

$N(1)=dN(0)+1$.

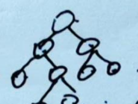
$N(h)=2^h+1+2+2^2+\cdots 2^{h-1}=2^{h+1}-1\ (h\geq -1)$

complete trees: not necessarily full.

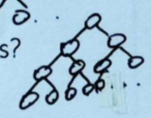
perfect trees are complete trees.

Practice

1) Draw a FULL tree with 9 nodes. Is it the same as your neighbor's?



2) Draw a COMPLETE tree with 12 nodes. Is it the same as your neighbor's?



3) Give an upper bound for the number of nodes in a tree of height h:

$O(c^h)$ $\quad N(h)=2^{h+1}-1$.

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

4) Give a lower bound for the height of a tree containing n nodes:

$\Omega(\log n)$ $\quad h(n)=\log_2(n+1)-1$.

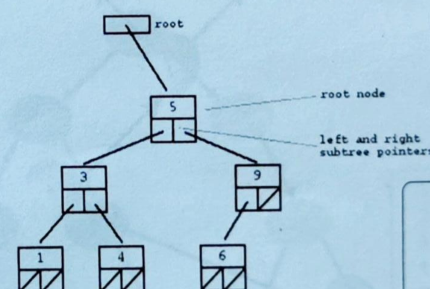
$=\lfloor \log_2 n\rfloor$ based on fact that heights are ints.

5) Give upper and lower bounds for the number of nodes in a complete tree of height h:

upper $c^{h+1}-1$ $\quad 2^{h+1}-1$

lower $c^{h+1}-c^h+1$. $\quad 2^{h+1}-2^h+1$

Rooted, directed, ordered, binary trees



Tree ADT:

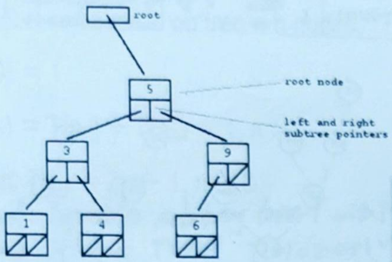
- insert
- remove
- traverse

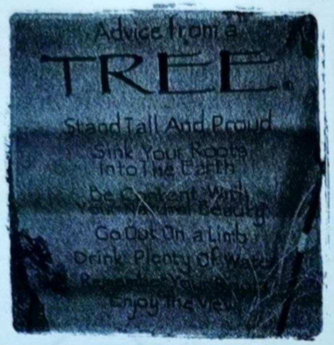
```
template<class T>
class tree {
public:
    …
private:
    struct Node {
        T data;
        Node * left;
        Node * left;
    };
    Node * root;
    …
};
```

Announcements:

HW2 due 2/14.

Rooted, directed, ordered, binary trees





Tree ADT:

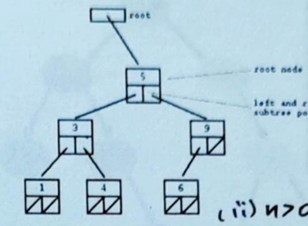
- insert
- remove
- traverse

```
template<class T>
class tree {
public:  // discuss later
private:
    struct Node {
        T data;
        Node * left;
        Node * right;
    };
    Node * root; // + more?
};
```

$N(n)$: # of nulls in our implementation of a tree of size n

Theorem: Our implementation of an n item binary tree has $n+1$ null pointers. $\forall n \geq 0$.

Consider an arbitrary binary tree implemented as on previous page. Let n denote the amount of data elements it holds.



(i) $n=0$ $T=\{\}$, we implement with one NULL pointer which is $0+1$. $N(0)=1$ ✓.

(ii) $n>0$ $T=\{root, T_R, T_L\}$

$|T|=n$, let $|T_L|=l$, $|T_R|=r$

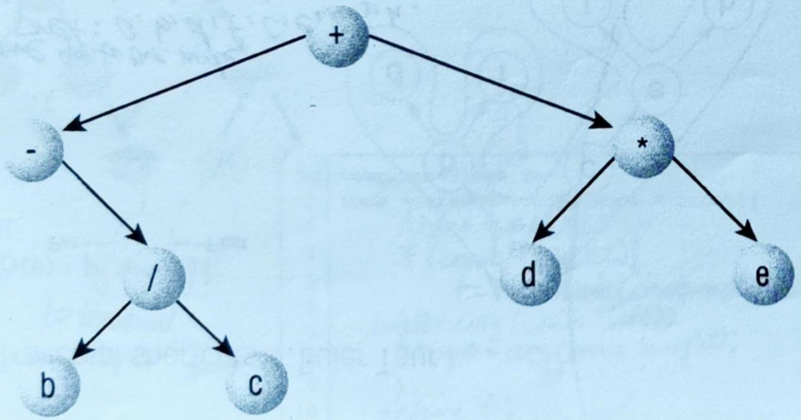
$n=l+r+1$

$N(n)=N(l)+N(r)$.

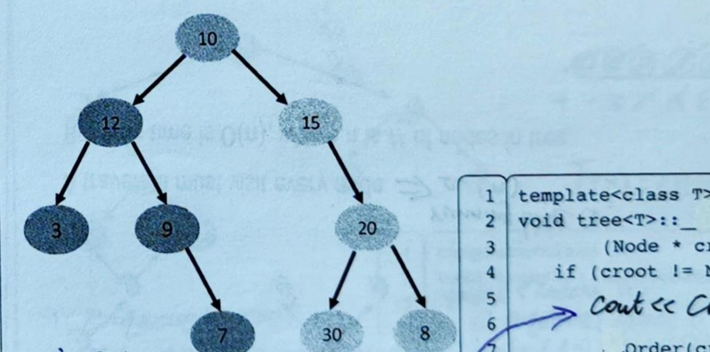
by IH $N(l)=l+1$ $N(r)=r+1$

$N(n)=N(l)+N(r) = l+1+r+1 = n+1$.

Traversal…a scheme for visiting every node.



Traversal…a scheme for visiting every node.



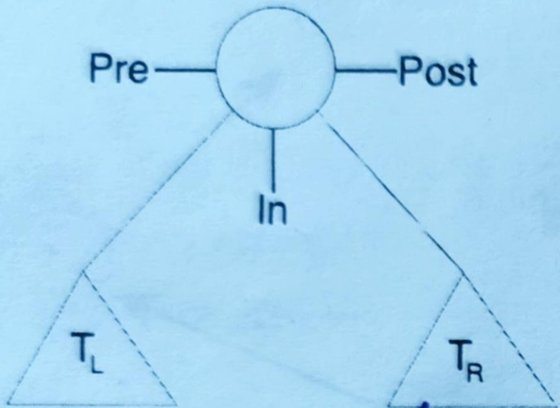
inOrder: 3, 12, 9, 7, 10, 15, 30, 20, 8

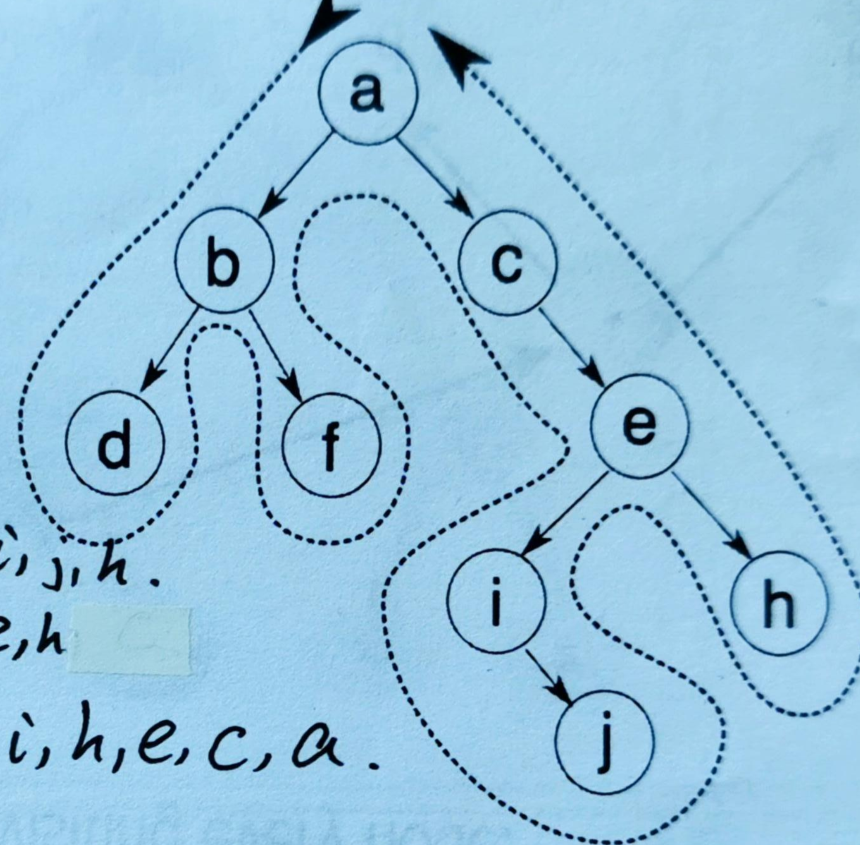
preOrder: 10, 12, 3, 9, 7, 15, 20, 30, 8.

postOrder: 3, 7, 9, 12, 30, 8, 20, 15, 10.

```
template<class T>
void tree<T>::___Order
      (Node * croot){
   if (croot != NULL){
      cout << croot->data;
      ___Order(croot->left);
      cout<<croot->data<<" ";
      ___Order(croot->right);
      cout<<croot->data;
   }
}
```

Traversal shortcuts...Euler Tour





first time got to the node
pre : a, b, d, f, c, e, i, j, h.

second time in : d, b, f, a, c, i, j, e, h

third time. post : d, f, b, j, i, h, e, c, a.

Announcements:

HW2 due 2/14.

Today: traversals, ADT dict, BST

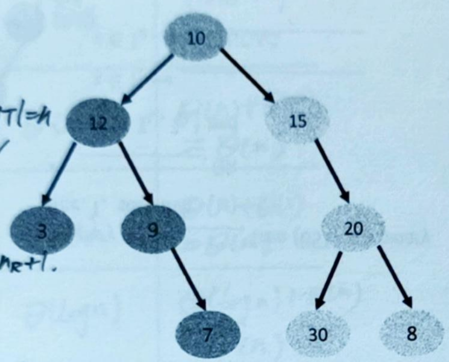
```
template<class T>
void tree<T>::preOrder
        (Node * croot){
   if (croot != NULL){
       cout << croot->data;
       preOrder(croot->left);
       preOrder(croot->right);
   }
}
```

T(n): ~~rt of traversal~~ evaluation of "if"s on tree with n nodes.

$T(0) = 1$

$T(n) = T(n_L) + T(n_R) + 1,\ n > 0.$

Thm: $T(n) = 2n + 1,\ n \geq 0.$



Proof. Consider arbitrary tree T with $|T|=n$

Base case: if $n=0$ $T(0)=1$ $T(0)=2\cdot 0+1$ ✓

if $n>0$, $T(n)=T(n_L)+T(n_R)+1$

$n=n_L+n_R+1$

IH ⇒ $T(n_L)=2n_L+1$ $T(n_R)=2n_R+1$.

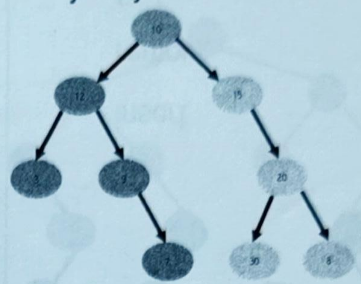
so $T(n)=2n_L+1+2n_R+1+1$

$=2(n_L+n_R+1)+1$

$=2n+1.$

$\Rightarrow T(n) \in \Theta(n).$

Mystery function: What's a good name?

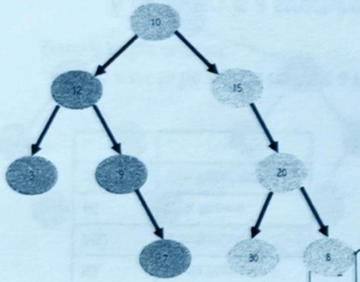


RT:

$\Theta(n)$ by analogy to postOrder traversal

```
template<class T>
void tree<T>::cleanUp (Node *& croot){
   if (croot != NULL) {
       cleanUp (croot->left);
       cleanUp (croot->right);
       delete croot;
       croot = NULL;
   }
}
```

Traversals: a broader view…

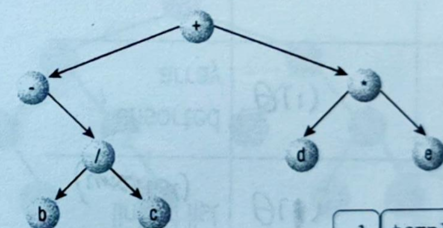


RT:

$\Theta(n)$ by analogy to traversal

```
template<class T>
Node * tree<T>::copy(Node * croot){
    Node* t = NULL;
    if (croot != NULL){
        t = new Node(croot->data);
    t->left = copy(croot->left);
    t->right = copy(croot->right);
    }
    return t;
}
```

Traversals: something totally different…



+ - * / d e b c.

$T(n) = \Theta(n)$

1. enqueue the root.
2. while queue is not empty
   a) deque a node.
   b) if node not Null
      i. print its data
      ii. enqueue its children

2n+1 iterations.

```
template<class T>
void tree<T>::levelOrder(Node* croot){
  queue<Node*> Q;
    Q.enqueue(croot);
    while (!Q.empty()){
      Node* t = Q.dequeue;
        if (t != NULL){
          cout << t->data;
          Q.enqueue(t->left);     Θ(1).   Θ(n).
          Q.enqueue(t->right);
        }
    }.
}
```

data<key, value>. Key: unique ID.

ADT Dictionary:

Suppose we have the following data…

ID #	Name
103	Jay Hathaway
92	Linda Stencel
330	Bonnie Cook
46	Rick Brown
124	Kim Petersen
…	…

…and we want to be able to retrieve a name, given a locker number.

More examples of key/value pairs:

CWL -> Advising Record

Course Number -> Schedule info

Color -> PNG

Vertex -> Set of incident edges

Flight number -> arrival information

URL -> html page

A *dictionary* is a structure supporting the following:

```
void insert(kType & k, dType & d)
void remove(kType & k)
dType find(kType & k)
```

Announcements:

HW2 due 2/14.

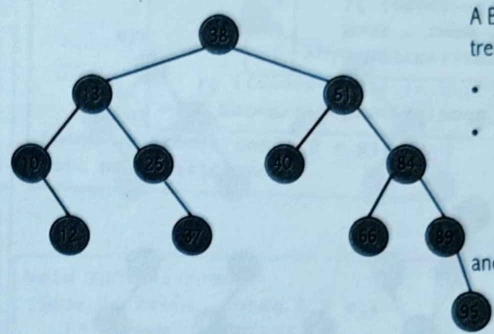
Today: BST

ADT Dictionary: Implementation options…

	insert	find	(find +) remove
linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)+\Theta(1) = \Theta(n)$
unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)+\Theta(1) = \Theta(n)$
sorted array	$\Theta(\log n)+\Theta(n)$ find open a spot	$\Theta(\log n)$	$\Theta(\log n)+\Theta(n) = \Theta(n)$.

Binary Search Tree

in-order traversal.



A Binary Search Tree (BST) is a binary tree, T, such that:

- T = { } empty
- T = {r, T_L, T_R} and

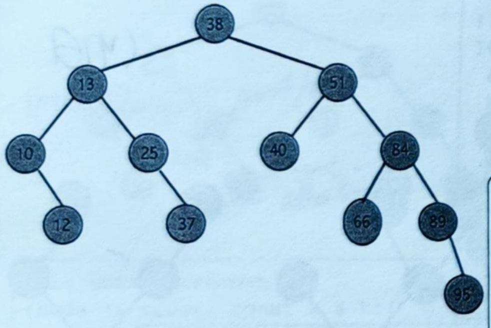
  $x \in T_L \rightarrow$ x.key < r.key.

  $x \in T_R \rightarrow$ x.key > r.key.

and $T_L + T_R$ are BST.

keys are unique.

Dictionary ADT: (BST implementation)



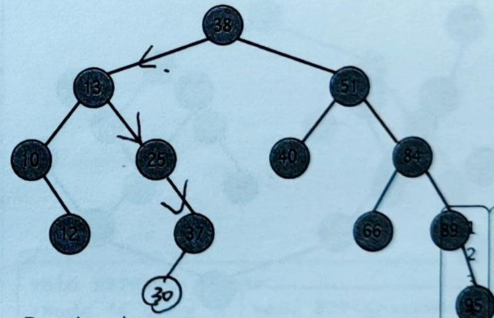
public. { insert, remove, find }

traverse $\Theta(n)$.

```
template<class K, class D>
class Dictionary{
public: //ctors etc
    …
private:
    struct Node {
        D data;
        K key;
        Node * left;
        Node * right;
    };
    Node * root;
    …
};
```

Binary Search Tree - Insert

insert(30);



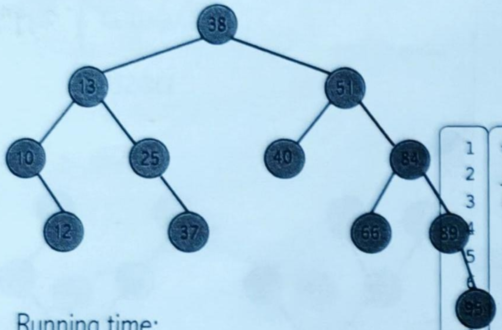
Running time:

Not like traversal

$\Theta(h)$ where h is height of tree.

```
template<class K>
void BST::insert
(Node * & croot, const K & key){

  if (croot == NULL)
     croot = new Node(key);


  else if (key < croot.key)
     insert(croot->left);

  else if (key > croot.key)
     insert(croot->right);
}
```

Binary Search Tree - Find




```
template<class K>
Node * Dictionary::find.
(Node * croot, const K & key){

   if (croot == NULL)
     return NULL;
   else if (croot->key == key)
     return croot;

   else if (croot->key < key)
     return find(croot->right, key);

   else
     return find(croot->left, key);
}
```

Running time:

$\Theta(h)$

Binary Search Tree - Remove

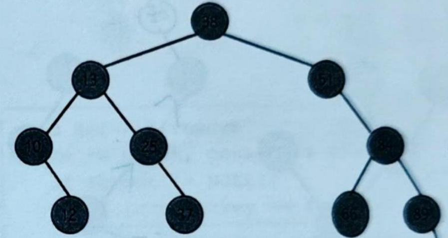
`T.remove(37);` ✓

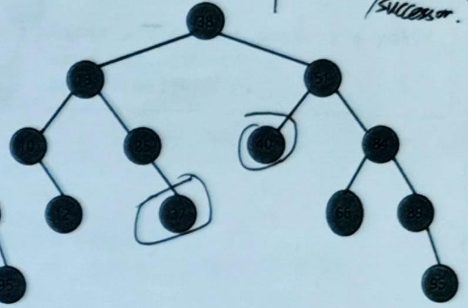
`T.remove(51);` ✓.

`T.remove(38);`

find largest in left or smallest in right.

in order precedor/successor.





https://cs221viz.netlify.com/src/bst

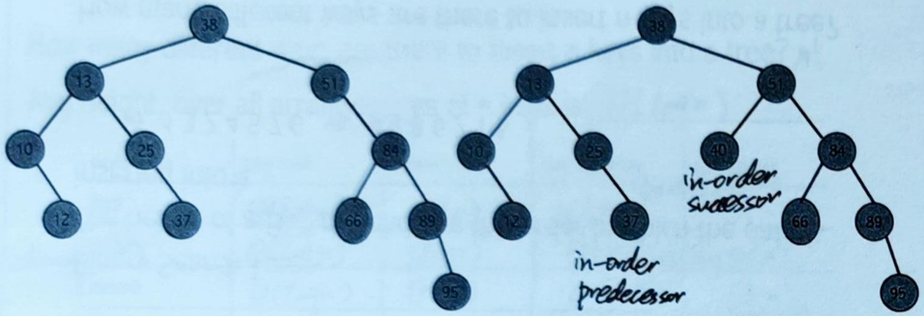
Announcements:

HW2 due 2/14.

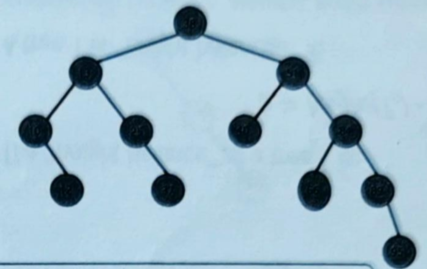
Today: BST removal + AVL intro https://cs221viz.netlify.com/src/bst

```
T.remove(37);

T.remove(51);
```

```
T.remove(38);
```



Binary Search Tree - Remove

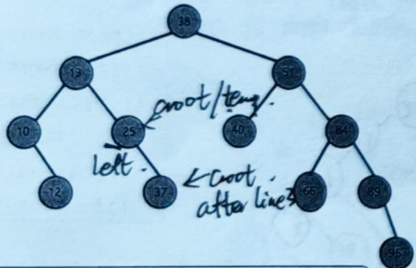


doRemoval(croot);
if (croot->key < key)
remove(croot->right);
else remove(croot->left);

```
void BST<K>::remove
(Node *& croot, const K & k){
    if (croot != NULL){
      if (croot->key == key)
   }
}
```

```
void BST<K>::doRemoval(Node * & cRoot) {
   if ((croot->left != NULL) && ((croot->right != NULL){
     twoChildRemove(cRoot);
   else
     zeroOneChildRemove(cRoot);
}
```

Binary Search Tree - Remove



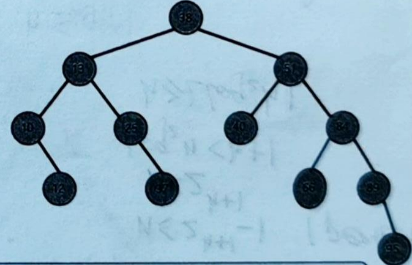

```
void BST<K>::remove
(Node *& croot, const K & k){
  if (cr
    if
    els
    els
    remove
  }
}
```

```
void BST<K>::doRemoval(Node * & cRoot) {
    if ((croot->left != NULL) && ((croot->right != NULL){
      el
}
```

```
1  void BST<K>::zeroOneChildRemove(Node * & cRoot) {
2      Node * temp = cRoot;
3      if (cRoot->left == NULL) cRoot = cRoot->right;
4      else cRoot = cRoot->left;
5      delete temp;
6  }
```

Binary Search Tree - Remove

running time: $\Theta(h)$
proportional to the height.



```
void BST<K>::remove
(Node *& croo
   if (cr
     if
     els
     els
   }
}
```

```
void BST<K>::twoChildRemove(Node * & cRoot) {
    Node * & iop = rightMostChild(cRoot->left);
    cRoot->key = iop->key;
    zeroOneChildRemove(iop);
}
```

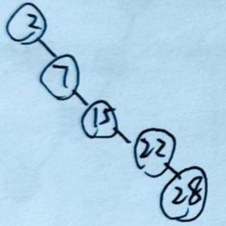
```
Node * & BST<K>::rightMostChild(Node * & cRoot) {
   if (cRoot->right == NULL) return cRoot;
   else return rightMostChild(cRoot->right);
}
```

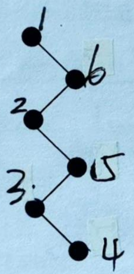
ADT *dictionary* supports the following:

```
void insert(kType & k, dType & d)
dType find(kType & k)
void remove(kType & k)
```

B<u>inary</u> S<u>earch</u> T<u>ree</u> implementation of a dictionary incurs running time <u>$\Theta(h)$</u> for all of these.

```
dict<int> myT;
myT.insert(2);
myT.insert(7);
myT.insert(15);
myT.insert(22);
myT.insert(28);
...
```





How many "bad" n-item trees are there?

n-1 R or L choices, 2^{n-1}

The *algorithms* on BST depend on the height (h) of the tree.

The *analysis* should be in terms of the amt of data (n) the tree contains.

So we need relationships between h and n:

$h \geq f(n)$ (what is max # of nodes in tree of height h?)

$n \leq 2^{h+1} - 1$ (perfect tree).

$n < 2^{h+1}$

$\log_2 n < h+1$

$h \geq \lfloor \log_2 n \rfloor$

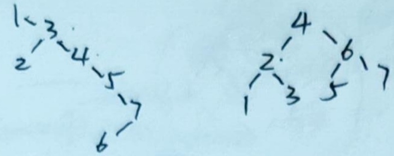
$h \leq g(n)$

$h \leq n-1$ ← room for improvement.

Good news, or bad?

Announcements:

PA2 due 03/09. MT2 03/04.



Today: AVL https://cs221viz.netlify.com/src/bst

The height of a BST depends on the order in which the data is inserted.

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

h=5 h=2

How many different ways are there to insert n keys into a tree? n!

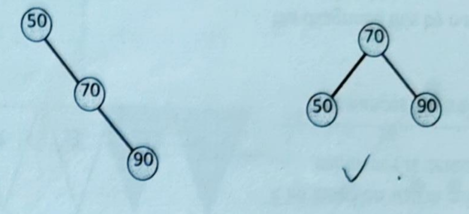
Avg height, over all arrangements of n keys is $\Theta(\log n)$.

operation	avg case (BST)	worst case (BST)	sorted array	sorted list
find	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$ binary search	$\Theta(n)$
insert	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$ shifting	$\Theta(n)$
delete	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
traverse	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Dictionary

---

something new… which tree makes you happiest?



The "height balance" of a (subtree) tree T is:

$$b = height(T_R) - height(T_L)$$

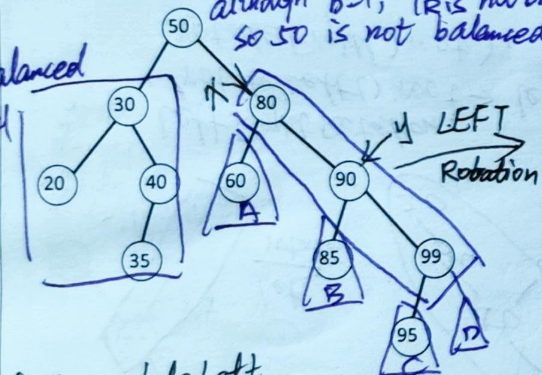
A tree T is "height balanced" if:

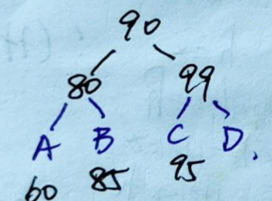
- $T = \{\}$ empty tree
- $T = \{r, T_L, T_R\}$ and $|b| \le 1$ and T_L & T_R are height balanced. if b=0 perfect tree.

---

operations on BST - rotations

although b=1, T_R is not balanced. so 50 is not balanced.





function rotateLeft

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

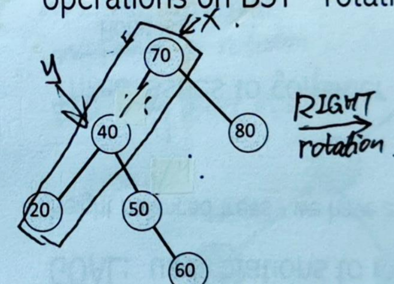
running time $\Theta(1)$

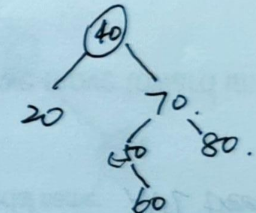
right rotation is analogous

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

---

operations on BST - rotations

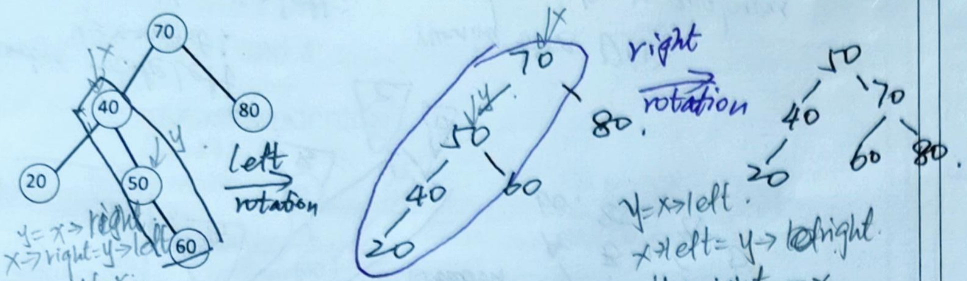




function rotateRight

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

operations on BST - rotations



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

y=x->left.
x->left=y->right.
y->right =x
x=y

LeftRightRotation
rotateLeft(croot->left);
rotateRight(croot);

balanced trees - rotations summary:

- there are 4 kinds: left, right, left-right, right-left (symmetric!)
- local operations (subtrees not affected)
- constant time operations
- BST characteristic maintained

GOAL: use rotations to maintain balance of BSTs.

height balanced trees - we have a special name: *AVL trees.*

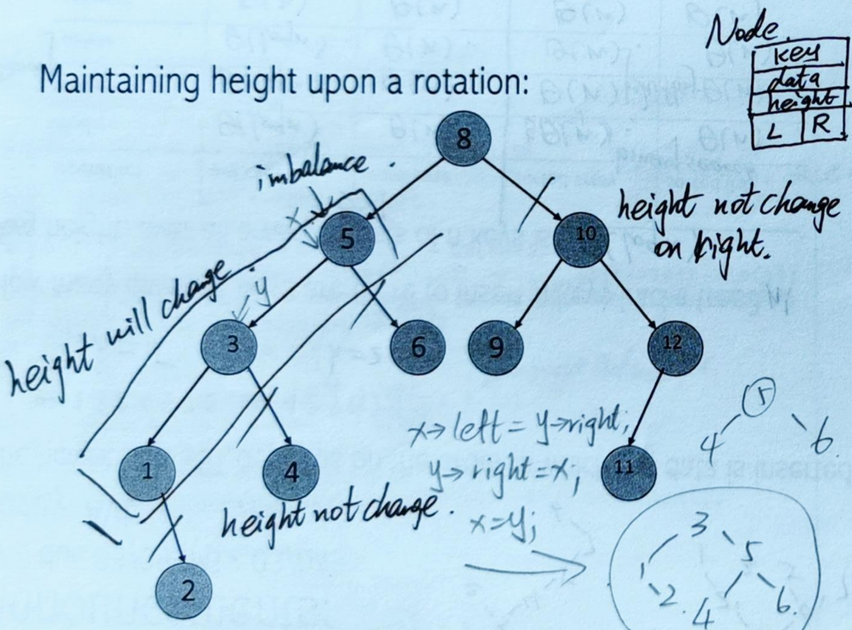
Three issues to consider as we move toward implementation:

Rotating

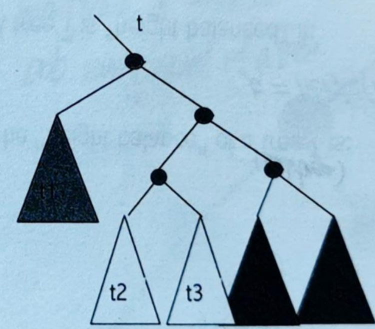
Maintaining height

Detecting imbalance *diagnosing rotations.*

Maintaining height upon a rotation:



AVL trees: rotations (identifying the need)



if an insertion was in subtrees t4 or t5, and if an imbalance is detected at t, then a ____________ rotation about t rebalances the tree.

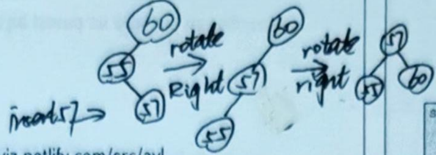
We diagnose this by noting that the balance factor at t→right is ______

If an insertion was in subtrees t2 or t3, and if an imbalance is detected at t, then a ______________________ rotation about t rebalances the tree.

We diiagnose this by noting that the balance factor at t→right is ______

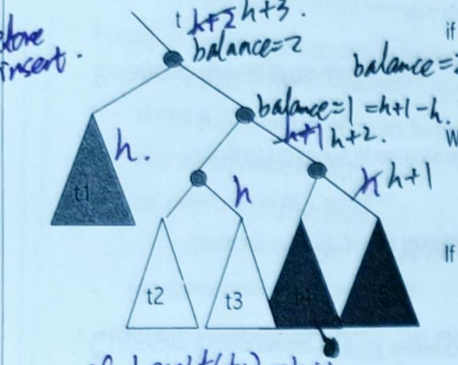
Announcements:

PA2 due 03/09. MT2 03/04.



Today: AVL insert and analysis https://cs221viz.netlify.com/src/avl

AVL trees: rotations (identifying the need)



if an insertion was in subtrees t4 or t5, and if an imbalance is detected at t, then a left rotation about t rebalances the tree. balance=2

We diagnose this by noting that the balance factor at t→right is 1. (h+1-h).

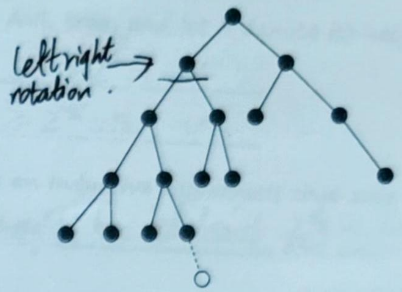
If an insertion was in subtrees t2 or t3, and if an imbalance is detected at t, then a right left rotation about t rebalances the tree.

if height(t1) =h+1 then adding one node to the right (making h+1 to h+2). will not change the structure!

We diagnose this by noting that the balance factor at t→right is -1 (h-(h+1))

AVL tree insert:

```
struct treeNode {
   T key;
   int height;
   treeNode * left;
   treeNode * right;
};
```



Insert:

- insert at proper place
- check for imbalance
- rotate if necessary
- update height

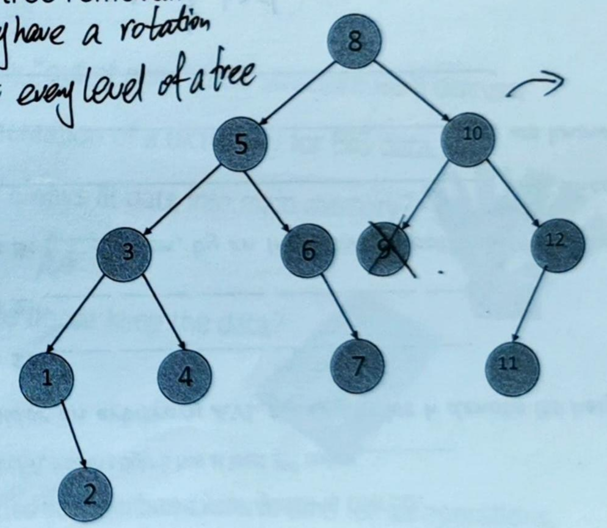
AVL tree insertions: time = Θ(n)

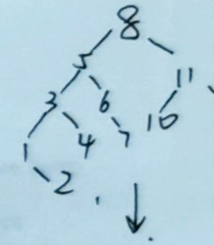
```
template <class T>
void AVLTree<T>::insert(const T & x, treeNode<T> * & t ){
   if( t == NULL ) t = new treeNode<T>( x, 0, NULL, NULL);   Base case
   else if( x < t->key ){   left
      insert( x, t->left );
      int balance = ht(t->rt) - ht(t->left);
      int leftBalance = ht(t->left->rt) - ht(t->left->left);
      if( balance == -2 )   // long on the left -> right rotation.
         if( leftBalance == -1 )
            rotate right ( t );
         else                          <- right Balance=-1.
            rotate left right ( t );}
   else if( x > t->key ){   right.
      insert( x, t->rt );
      int balance = ht(t->rt) - ht(t->left);
      int rtBalance = ht(t->rt->rt) - ht(t->rt->left);
      if( balance == 2 )
         if( rtBalance == 1 )
            rotate left ( t );
         else
            rotate right left. ( t );
   }
   t->ht=max(ht(t->left ), ht(t->rt))+ 1;}
```

Θ(1)

AVL tree removal:

may have a rotation at every level of a tree





Announcements:

PA2 due 03/09. MT2 03/04.

Today: AVL analysis, BTrees https://cs221viz.netlify.com/src/avl we already know h is $\Omega(\log n)$

AVL tree analysis:

Since running times for Insert, Remove and Find are O(h), we'll argue that h = O($\log n$).

Putting an upper bound on the height for a tree of n nodes is the same as putting a lower bound on the number of nodes in a tree of height h.

$N(-1) = 0$
$N(0) = 1$
$N(1) = 2$.

- Define N(h): least # of nodes in AVL tree of ht h
- Find a recurrence for N(h): N(h) = 1 + N($h-1$) + N ($h-2$)
- We simplify the recurrence:

  $N(h) \geq 2N(h-2)$

- Solve the recurrence: (guess a closed form)

  $N(h) = 2^{\frac{h}{2}}$ $h \geq 0$

AVL tree analysis: prove your guess is correct.

Thm: An AVL tree of height h has at least $2^{h/2}$ nodes, $h \geq 0$.

Consider an arbitrary AVL tree, and let h denote its height.

Case 1: $h=0$ $N(0)=1 \geq 2^{\frac{0}{2}}=1$ ✓

Case 2: $h=1$ $N(1)=2 \geq 2^{\frac{1}{2}}=\sqrt{2}$ ✓.

Case 3: $h>1$ then, by an Inductive Hypothesis that says $\forall j<h$, an AVL tree of height j has at least $2^{\frac{j}{2}}$ nodes, and since $N(h) \geq 2N(h-2)$, we know that $N(h) \geq 2 \cdot 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}}$.

Let n denote the actual # of nodes of an AVL tree of height h.

Punchline: $n \geq N(h)$ $n \geq 2^{\frac{h}{2}}$ $h \geq 0$

$h \leq 2\log_2 n$, $n \geq 1$ $h = O(\log n)$.

Classic balanced BST structures:

- Red-Black trees – max ht $2\log_2 n$.

  Constant # of rotations for insert, remove, find.

- AVL trees – max ht $1.44\log_2 n$.

  O(log n) rotations upon remove. no rotation on a find. 1 rotation on insert. O(log n) rotation on remove.

Balanced BSTs, pros and cons:

- Pros:
  - Insert, Remove, and Find are always O(log n)
  - An improvement over: BST, linked list, arrays
  - Range finding & nearest neighbor
- Cons:
  - Possible to search for single keys faster via hashing
  - If data is so big that it doesn't fit in memory it must be stored on disk and we require a different structure.

B-Trees

The only "out of core" data structure we'll discuss.

Implementation of a dictionary for BIG data.

Can we always fit data into main memory?

No.

So where do we keep the data?

Big-O analysis assumes uniform time for all operations.

But… remote data access is slower.

The Story on Disks

2GHz machine gives around 2m instructions per millisecond.

Seek time around 5ms for a current hard disk.

Imagine an AVL tree storing North American driving records.

How many records? 200,000,000

How much data, in total? 200m × 1 MB = 2 TB.

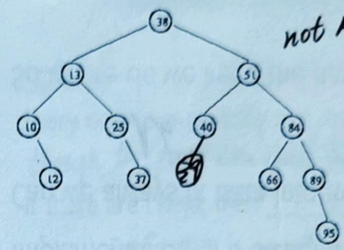
How deep is the AVL tree? $\log_2 200m \approx 56$.

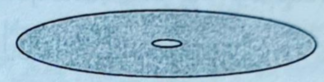
How many disk seeks to find a record?

The Story on Disks, continued.

Suppose we weren't careful…

not AVL, add 39.





B Tree of order m

12	18	27	52	58	63	77	89

B tree node.

contains $[\frac{m}{2}, m-1]$ keys.

except the root.

(which may have only 1 key)

has capacity m-1.

Goal: Minimize the number of reads from disk

- Build a tree that uses 1 disk block per node
  - Disk block is the fundamental unit of transfer
- Nodes will have more than 1 key
- Tree should be balanced and shallow
  - In practice branching factors over 1000 often used

491 709

205 280 300 | 517 620. | 882 929 . 931. 995.

250.

insert 250.

insert 999.

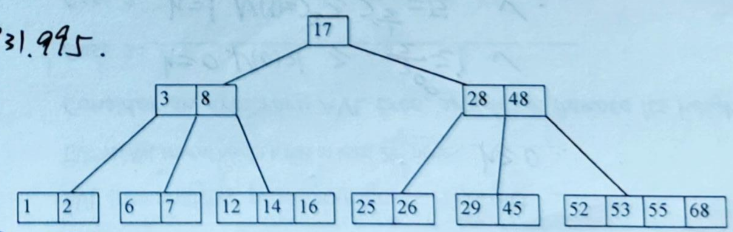
⇒ 491 709 931

205 250 280 300 | 517 620 | 882 929 | 995 999

Definition of a B-tree

B-tree of order *m* is an *m*-way tree

- For an internal node, # keys = #children -1
- All leaves are on the same level
- All leaves hold no more than *m*-1 keys
- All non-root internal nodes have between $\lceil m/2 \rceil$ and m children
- Root can be a leaf or have between 2 and *m* children.
- Keys in a node are ordered.



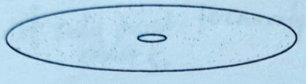
Announcements:

PA2 due 03/09. MT2 03/04.

Today: BTrees

BTree of order m

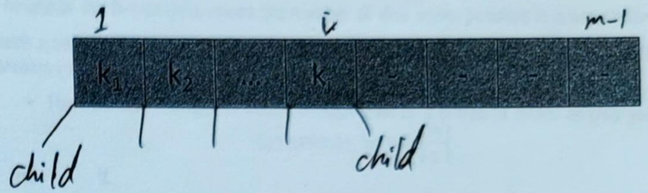
12	18	27	52	58	63	77	89



Goal: Minimize the number of reads from disk or remote sources.

- Build a tree that uses 1 disk block per node
  - Disk block is the fundamental unit of transfer
- Nodes will have more than 1 key
- Tree should be balanced and shallow
  - In practice, m over 1000 often used!!

B Tree of order m: Nodes



Every node has capacity m-1, but every node contains at least $\lceil \frac{m}{2} \rceil - 1$ keys. though the root can hold fewer.

If there are i keys, then ______ $i+1$ ______ child pointers.

$i \in [\lceil \frac{m}{2} \rceil - 1, m-1]$

least # of children for a non-leaf node: $\lceil \frac{m}{2} \rceil$

greatest #: m.

May also maintain sibling pointers.

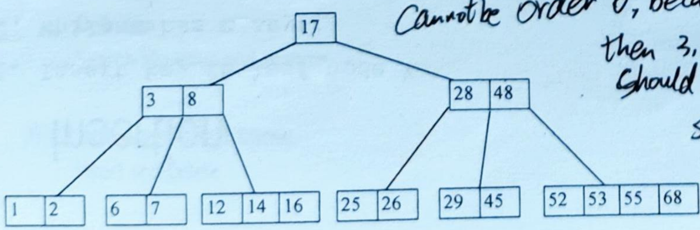
Level order pointers, expedits removal.

Note: root & leaves have slightly different constraints.

Definition

B-tree of order m is an m-way tree

- For an internal non-leaf node, # keys = #children -1
- All leaves are on the same level
- All leaves hold no more than m-1 keys
- All non-root internal nodes have between $\lceil m/2 \rceil$ and m children
- Root can be a leaf or have between 2 and m children.
- Keys in a node are ordered.



Cannot be order 6, because. if m=6, then 3, 8, 17, 28, 48 should be in one node.

so order can only be 5.

Search

```
bool B-TREE-SEARCH(BtreeNode & x, T key){
   int i = 0;
   while ((i < x.numkeys) && (key > x.key[i]))
      i++;
   if ((i < x.numkeys) && (key == x.key[i]))
      return true;
   if (x.leaf == true)
      return false;
   else{
      BtreeNode b=DISK-READ(x.child[i]);
      return B-TREE-SEARCH(b,key);
   }
}
```

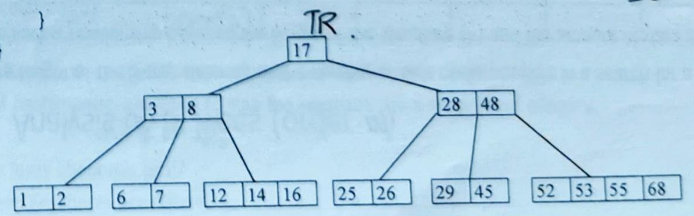
Using binary search here. change to $\Theta(\log m)$. from $\Theta(m)$. but does not change too much, since disk-read is dominant factor.

find the correct child i

key is found.

x is at leaf and does not have the key.

Search(TR, 55)



Insertion

1. Insert key in leaf node X
2. While X has m keys:

    Split X into two nodes:

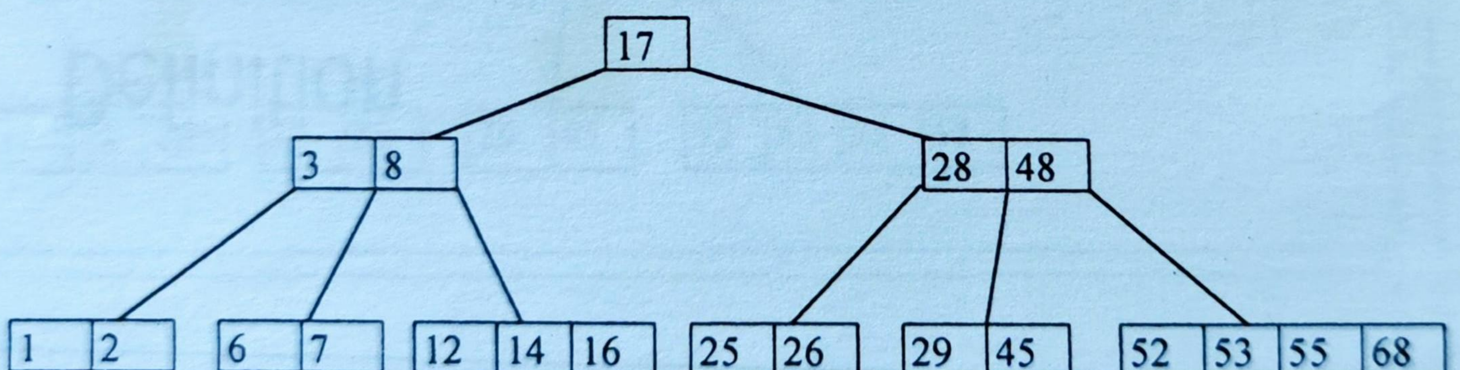
        orig has m/2 smallest keys

        new has m/2 largest keys

    Insert middle key into parent & attach new node

    (If X is root, create new node and attach both)

    Set X to be the parent.



Announcements:

PA2 due 03/09.

Today: finish Btrees, start something new!!

B-tree of order *m* is an *m*-way tree

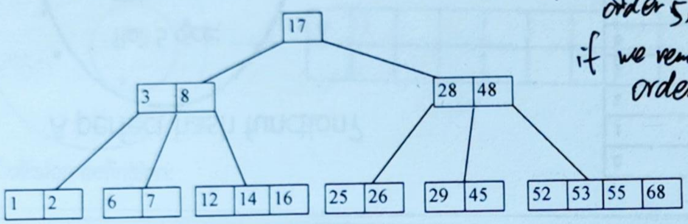
height of B-tree: $\log_m \frac{n+1}{2} = O(\log_m n)$

disk seek = height.

- For an internal node, # keys = #children -1
- All leaves are on the same level
- All leaves hold no more than *m*-1 keys
- All non-root internal nodes have between $\lceil m/2 \rceil$ and m children
- Root can be a leaf or have between 2 and *m* children.
- Keys in a node are ordered.

if only insert, it can only be order 5,

if we remove, it maybe order 6.



Analysis of B-Trees (order *m*)

The height of the B-tree determines the number of disk seeks possible in a search for a key.

We seek a relationship between the height of the structure (h) and the amount of data it contains (n).

- The minimum number of *nodes* in each level of a B-tree of order *m*. (For your convenience, let $t = \lceil \frac{m}{2} \rceil$

root	1
level 1	2
level 2	2t
level h	$2t^{h-1}$

- The total number of nodes is the sum of these:

$$1 + 2\sum_{i=0}^{h-1} t^i = 1 + 2 \cdot \frac{t^h - 1}{t-1}$$

- So, the least **total** number of *keys* is:

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

$n \geq 2t^h - 1$.

$h \leq \log_t \frac{1}{2}(n+1)$.

$h = O(\log_t n)$

$= O(\log n)$

since m (t) is constant.

Summary

B-Tree search: $O(\log m)$ instead with binary search.

O(m) time per node to find key or appropriate child

$O(\log_m n)$ height implies $O(m \log_m n)$ total time

BUT: m is constant. search is $O(\log n)$.

Insert and Delete have similar stories.

What you should know:

Motivation

Definition

Search algorithm and analysis

What you should not know:

Insert and Delete

Hashing - using "hash tables" to implement dictionaries.

Structure of a dictionary:

Key -> Value

Locker # -> student

Course Number -> Schedule info

Vertex -> Set of incident edges

URL -> html page

dice roll -> payoff amt

Associative Array:

Dictionary with a particular interface

Overloads operator[] for insert and find

```
myDict["Miguel"] = 22;
int d  = myDict["Miguel"];
```

0	1	2					
		arr[2]					

(defn) Keyspace – a (math) description of the keys for a set of data. K

Goal of hashing: use a function to map the keyspace into a small set of integers.

$h: K \to \mathbb{Z}_0$.

What's fuzzy about this goal?

Problem: Keyspaces are often large…

$= O(\log n)$
since $m(t)$ is
constant.

Hashing - using "hash tables" to implement *dictionaries*.

Structure of a dictionary:

Key -> Value

Locker # -> student

Course Number -> Schedule info

Vertex -> Set of incident edges

URL -> html page

dice roll -> payoff amt

Associative Array:

Dictionary with a particular interface

Overloads operator[] for insert and find

```
myDict["Miguel"] = 22;
int d  = myDict["Miguel"];
```

0	1	2					
		arr[2]					

(defn) Keyspace – a (math) description of the keys for a set of data. K

Goal of hashing: use a function to map the keyspace into a small set of integers.

$h: K \to \mathbb{Z}_0$.

What's fuzzy about this goal?

Problem: Keyspaces are often large…

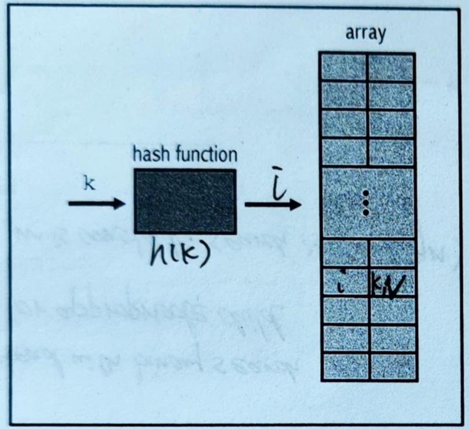
Overview:

client code

declares an object of ADT dictionary
```
dict<ktype, vtype> d;
```
d.operator[](k);

ex: insert is `d[k] = v;`

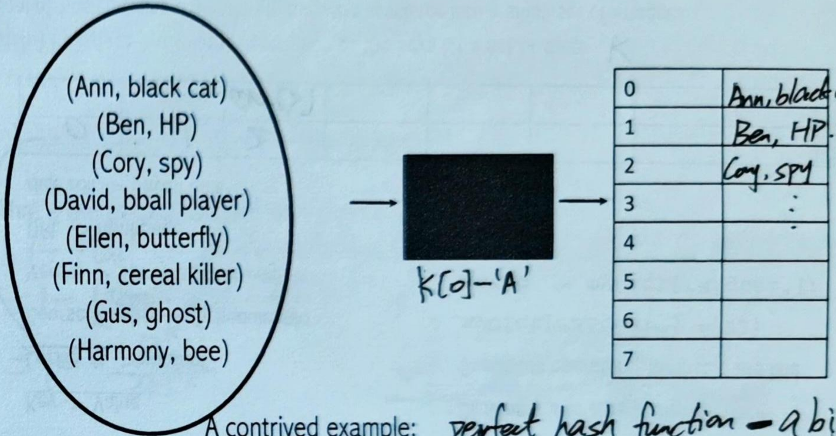
class `dict` (implemented with a hash table)



A Hash Table consists of:

- hash function.
- an array
- CRS

A perfect hash function:



A contrived example: perfect hash function — a bijecti

these keys have a fabulous hash fn.

a. each key hashes to a different int injective.

b. collection of keys hash to a seq of ints surjectivity.

Our purpose is to investigate general purpose hash function.

Announcements:

PA2 due today! PA3 out soon!

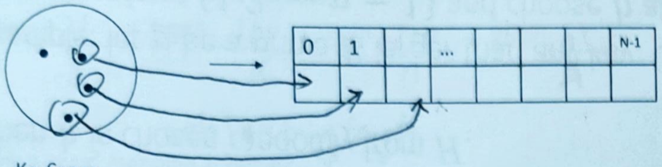
Today: Hash functions, continued

General Purpose Hash Functions

What characterizes a good hash function?

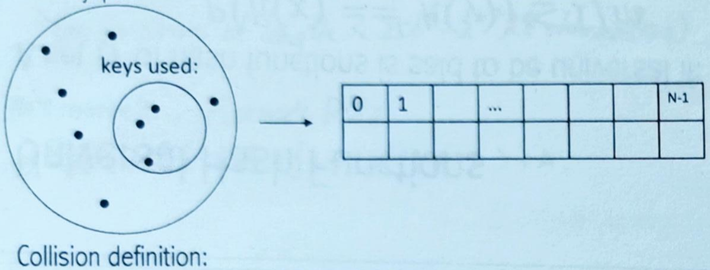
- $h(k)$ computes in $\Theta(1)$
- $k_1 = k_2 \Rightarrow h(k_1) = h(k_2)$
- $k_1 \neq k_2$ $P(h(k_1) == h(k_2)) = \frac{1}{N}$ where N is the table size.

KeySpace



Easy, if |KeySpace| ~ N

KeySpace



Collision definition:

Probability

$P(\text{one random students birthday is not today}) = \frac{364}{365} = 1 - \frac{1}{365}$.

What's the probability that someone in this room has a birthday today?

$1 - P(\text{no one has a birthday today})$

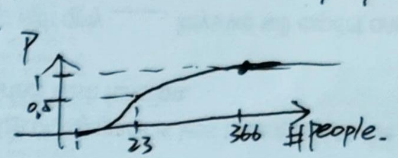
$= 1 - \left(\frac{364}{365}\right)^{110}$ ← # of people.

What's the probability that there are 2 people with the same birthday?

$1 - P(\text{all on different days})$. k is the number of people in the room

$= 1 - \left(\frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365}\right) \cdots$

$= 1 - \frac{P_{365}^{k}}{365^k}$



Devise and analyze an algorithm to find the people with the same bday...

index on days and place people on table.

Expected Value

discrete, finite. weighted average of possible values.

How many pips do we expect to see on a die?

Definitions that will help us:

X: # of pips. $x \in \{1,2,3,4,5,6\}$.

E[X]: $\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \cdots + \frac{1}{6} \cdot 6 = 3.5$.

$E[X] = \sum_{x \in X} x P(X=x)$

How many pips do we expect to see on 2 dice?

X, Y: independent dice.

$E[X+Y] = \frac{1}{36} \cdot 2 + \frac{2}{36} \cdot 3 + \cdots \frac{1}{36} \cdot 12 = 7$

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Linearity of Expectation:

$E[X+Y] = E[X] + E[Y]$

Expected # of collisions

What is expected # of people who share a bday w someone else?

Definitions that will help us:

Xij: 1 if student i has the same birthday as student j, 0 otherwise.

E[Xij]: $P(x_i \text{ and } x_j \text{ has same birthday}) \cdot 1 + P(\cdots) \cdot 0 =$ first term.

Sum over all Xij are the number of shared bdays:

X: $\sum_{i=0}^{k-1} \sum_{j=i+1}^{k} x_{ij}$.

$E[X] = \sum_{i=0}^{k-1} \sum_{j=i+1}^{k} E[X_{ij}] = \sum_{i=0}^{k-1} \sum_{j=i+1}^{k} P(i \,\&\, j \text{ have the same birthday})$.

$= \frac{1}{365} \sum_{i=0}^{k-1} \sum_{j=i+1}^{k} 1$.

$= \frac{1}{365} \cdot \frac{k(k-1)}{2}$

Collisions: if we randomly put k items into m bins, we expect $\frac{1}{N} \cdot \frac{k(k-1)}{2}$ pairs to collide. N is table size.

Implication: $\frac{1}{N} \cdot \frac{k(k-1)}{2} \geq 1$

so if # keys $k \geq \sqrt{2N}$ there will be a collision.

Announcements:

PA3 out soon!

Today: Hash functions, continued

Mapping strings to integers

How can we map 'Andy' to a number?

$256^3 \underset{y}{\underline{121}} + 256^2 \underset{d}{\underline{160}} + 256^1 \underset{n}{\underline{110}} + 256^0 \underset{A}{\underline{65}}$

$256^4 \times$ string length .

Is this a good mapping scheme?

Yes for strings of length $< 256^8 = 2^{64}$ (8 characters).

We'll rewrite it... Horner's Rule.

$256(256(256(y) + d) + n) + A.$

Hashing Strings (an example)

Given: 8 character strings are easy to hash

The idea: Select 8 random positions from long strings and hash that substring.

A bunch of strings:

```
Lookyhere, Huck, being rich ain't going
No! Oh, good-licks; are you in real dead
Just as dead earnest as I'm sitting here
nto the gang if you ain't respectable, y
Can't let me in, Tom? Didn't you let me
Yes, but that's different. A robber is m
irate is -- as a general thing. In most
Now, Tom, hain't you always ben friendly
ut, would you, Tom? You wouldn't do that
Huck, I wouldn't want to, and I DON'T wa
ay? Why, they'd say, 'Mph! Tom Sawyer's
t!' They'd mean you, Huck. You wouldn't
uck was silent for some time, engaged in
Well, I'll go back to the widder for a m
can come to stand it, if you'll let me
All right, Huck, it's a whiz! Come along
Will you, Tom -- now will you? That's go
he roughest things, I'll smoke private a
hrough or bust. When you going to start
Oh, right off. We'll get the boys togeth
```

Hashing Strings (an example)

Given: 8 character strings are easy to hash

The idea: Select 8 random positions from long strings and hash that substring.

A bunch of strings:

```
http://en.wikipedia.org/wiki/Le%C5%9Bna_Grobla
http://en.wikipedia.org/wiki/Blow_the_Man_Down
http://en.wikipedia.org/wiki/Swen_K%C3%B6nig
http://en.wikipedia.org/wiki/2/7th_Cavalry_Commando_Regiment_(Australia)
http://en.wikipedia.org/wiki/Salman_Ebrahim_Mohamed_Ali_Al_Khalifa
http://en.wikipedia.org/wiki/Alice_High_School
http://en.wikipedia.org/wiki/Beautiful,_Dirty,_Rich
http://en.wikipedia.org/wiki/RFA_Sir_Bedivere_(L3004)
http://en.wikipedia.org/wiki/Birthright_(band)
http://en.wikipedia.org/wiki/Jacky_Vimond
http://en.wikipedia.org/wiki/Vachon
http://en.wikipedia.org/wiki/McCarthy_%26_Stone
http://en.wikipedia.org/wiki/Salisbury,_New_Hampshire
http://en.wikipedia.org/wiki/A_Line_of_Deathless_Kings
http://en.wikipedia.org/wiki/Newfoundland_Irish
http://en.wikipedia.org/wiki/Beatrice_Politi
http://en.wikipedia.org/wiki/Bona_Sijabat
http://en.wikipedia.org/wiki/Sour_sanding
http://en.wikipedia.org/wiki/Dr_Manmohan_Singh_Scholarship
http://en.wikipedia.org/wiki/Religion_in_Jordan
```

Hashing strings

```
int hash ( string s, int p) {
  int h = 0;
  for (i = s.length-1; i >= 0; i--)
     h = (256 * h + s[i]) % p;
  return h;
```

Horner's Rule.

Running time?

Problem: suppose ascii(A)=61

hash('A', 61); $= 0$.

hash('AA', 61); $= (256 \times 61 + 61) \% 61 = 0$.

hash('AAA', 61); $= (256^2 \cdot 61 + 256 \cdot 61 + 61) \% 61 = 0$.

Solution:

keep hash function secret by creating a random hash function.

Universal Hash Functions

A set H of hash functions is said to be universal if:

$$P\big(h(x) == h(y)\big) \leq \frac{1}{m},$$ m is the table size. (satisfies SUHA)

When h is chosen randomly from H.

Example: let p be a prime # larger than any key.

- Choose a at random from $\{1,2, \ldots\ p-1\}$ and
- choose b at random from $\{0,1, \ldots\ p-1\}$ then let

$$h(x) = ((ax + b) \bmod p) \bmod m$$

key

Find the largest set of keys that collide:

① h(x) = (3x + 2) mod 9

x=0 x=1 x=2 .

[2, 5, 8]

② h(x) = (3x + 2) mod 11 explores the entire space.

x=0 x=1 x=2 x=3 x=4 x=5 x=6 x=7. x=8 x=9 x=10 x=1.

2 5 8 0 3 6 9 1 , 4, 7, 10, 2 .

Observations: ① spreads out the data. ② can still map infinitely many keys to the same int.

Hash function summary

We can only avoid collisions if the size of keyspace is less than equal to the size of our hash table and we have a perfect hash function.

m is the table size.

We have to deal with collisions: even with only $\sqrt{2m}$ keys we will expect one.

if m=10,000, we expect a collision with only 140 keys.

We need a collision resolution strategy

- Separate chaining
- Open addressing
- Other strategies we won't talk about

Collision resolution –

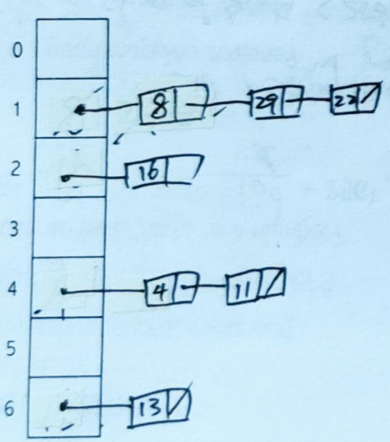
SUHA: simple universal hashing assumption

Separate Chaining: (an example of open hashing) allow collisions.

S = {16, 8, 4, 13, 29, 11, 22}

|S| = n h(k) = k%7

\# of keys



Load factor:

$$\alpha = \frac{n}{m} \left(\frac{\#\text{ keys}}{\text{table size}}\right)$$

	Worst case	Under SUHA
Insert	$\Theta(n)$	α
Remove/find	$\Theta(n)$	α .

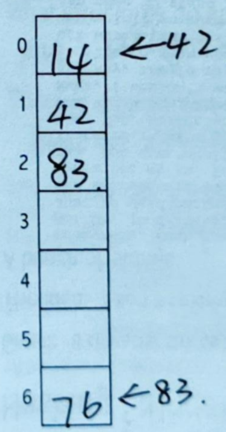
keys spread out evenly

$\alpha \geq 0$. unconstraint. under separate chaining.

Collision Handling - Probe based hashing: (example of closed hashing)

open addressing

S = {76, 14, 42, 83, 11, 22} |S| = n h(k) = k%7



\# of failed attempts

Try H(k,0) = (h(k) + 0) % 7. If full...

try H(k,1) = (h(k) + 1) % 7. If full...

try H(k,2) = (h(k) + 2) % 7. If full...

try...

find: keep looking until an empty block.

if remove(42) then find(83), it cannot find it.

— use flag { never occupied 0 insert; once occupied, now empty 1; now occupied. 2 }

find.

https://cs221viz.netlify.com/src/linearhashing.html

Announcements:

PA3 out, due 03/30!

Today: Hash functions, fin. Something new!! https://cs221viz.netlify.com/src/linearhashing.html

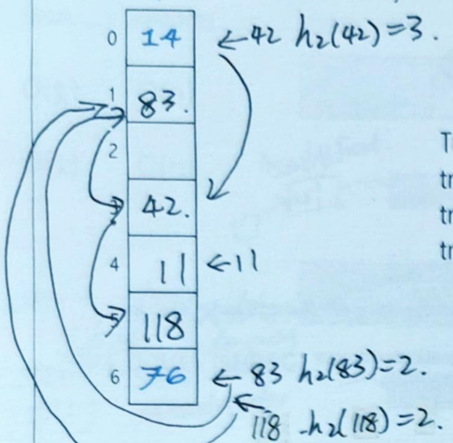
Collision Handling - Probe based hashing: (another example of closed hashing)

Double hashing (two hash functions)

$S = \{76, 14, 42, 83, 11, 22\}$ $|S| = n$ $H(k,i) = h_1(k) + i*h_2(k) \% P$

$h_1(k) = k \% 7$ init_hash

$h_2(k) = (5 - k\%5) \to q.$ step 5



Try $H(k,0) = (h(k) + 0*h_2(k)) \% 7$. If full...

try $H(k,1) = (h(k) + 1*h_2(k)) \% 7$. If full...

try $H(k,2) = (h(k) + 2*h_2(k)) \% 7$. If full...

try...

p, q are primes.

p is the table size.

$2 < q < p$.

Hash table performance: expected # of probes for Find(key) under SUHA

Linear probing -

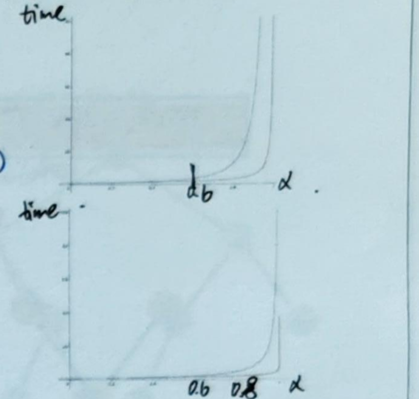
successful: $\frac{1}{2}(1 + 1/(1-\alpha))$

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: $1 + \alpha$

Do not memorize!

Observe: $\alpha = \frac{n}{m}$.

α can be greater than 1

- As α increases, running times increase!
- If α is held constant then running times are constant!

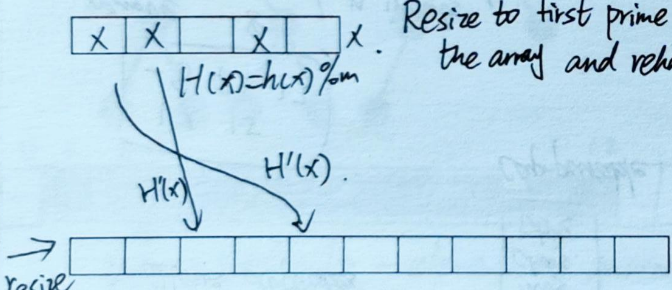
ReHashing:

hold α constant if we control table size m.

What if the array fills? Double the array and copy the data.

what if the load factor exceeds 0.6?

Resize to first prime bigger than double the array and rehash the data.



Hashing Miscellaneous Discussion –

Which collision resolution strategy is better?

$\alpha > 1$: separate chaining.

caching : probe based strategies

What structures do hash tables replace for us? Dictionaries

AVL : $\Theta(\log n)$ for worst and avg time. Hash Table : $\Theta(n)$ worst case

$\Theta(1)$ on average amortized.

There is a constraint on Keyspaces for BST that does not affect hashing...

Keys for AVL/BST must be comparable

Why do we talk about balanced BST if hashing is so great?

Comparision based structures allow for search when exact keys aren't known – max, nearest neighbor.

More resources:

http://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf

http://en.wikipedia.org/wiki/Hash_function

Applications of hashing?

Area of active research in mathematics to develop general purpose hash functions.

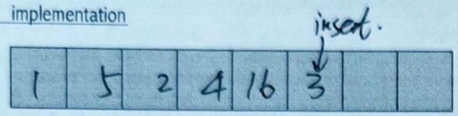
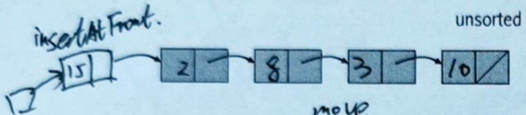
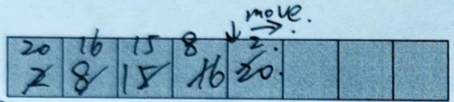
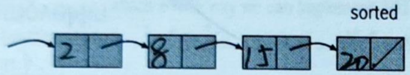
all collision resolution algorithms give the same worst case time complexity for find

Announcements:

PA3 out, due 03/30!

Priority Queue ADT:

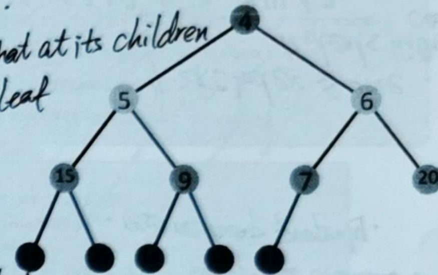
insert → comparable keys
remove min → key
key – priorities

insert	removeMin	implementation
O(1)	O(n)	
O(1)	O(n)	unsorted 
~~O(log n)~~ O(n) shift data.	O(1) sort data in decreasing order	
~~O(log n)~~ O(n)	O(1) increasing order	sorted 

Priority Queue: another implementation option

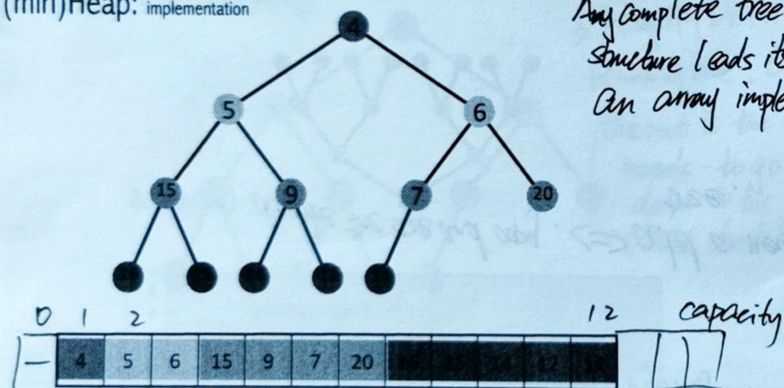
Tell me everything you can about this structure:

- binary tree – complete.
- each node's key is ≤ that at its children
- every path from root to leaf is non-decreasing



Heap: min element is at the top.

(min)Heap: implementation



Any complete tree-based structure leads itself to an array implementation.

Keys in level order – elements of an array.

LeftChild(i) = at index 2i.

RightChild(i) = 2i+1.

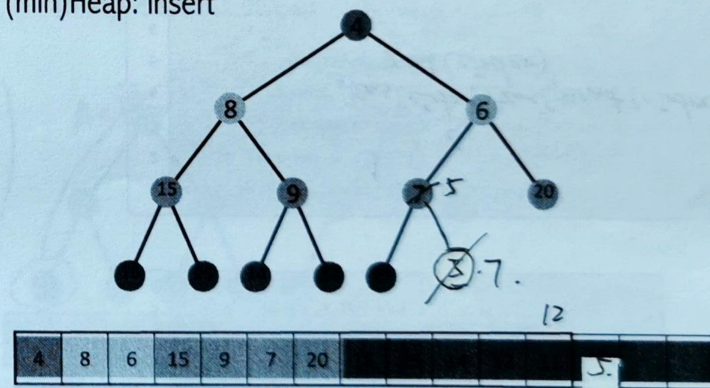
parent(i) = $\lfloor \frac{i}{2} \rfloor$

How tall is a heap of size n?

$h \le \lfloor \log_2 n \rfloor$.

Priority Queue.

(min)Heap: insert



insert at end.

bubble it up ("heapify up")

https://cs221viz.netlify.com/src/heap.html

Running time: O(log n).

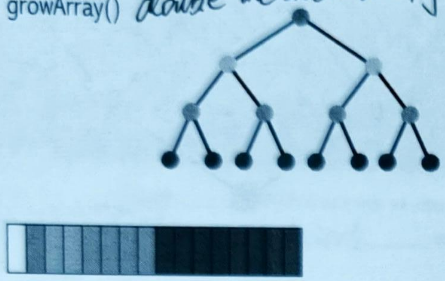
Constant time to insert at end.

heapify up is proportional to the height.

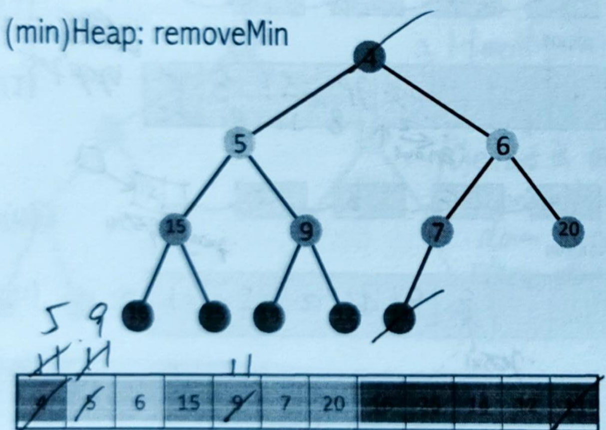
Code:

```
void heap<t>::insert(const T & key){
        if (size == capacity) growArray(); // array is full
        size++;
        items[size] = key;
        heapifyUp(size);
}
```

growArray() double the size and copy. ⟺ add a new level to the tree.



(min)Heap: removeMin



Swap last key to root
restore order by swapping downward.

Code:

```
void heap<t>::insert(const T & key){
        if (size == capacity) growArray();
        size++;
        items[size] = key;
        heapifyUp(size);
}
```

```
void heap<t>::heapifyUp(int cIndex){
   if (cIndex > 1)
      if (items[cIndex] < items[parent(cIndex)]){
         swap(items[cIndex], items[parent(cIndex)]).
         heapifyUp(parent(cIndex))
      }
}
```

Code:

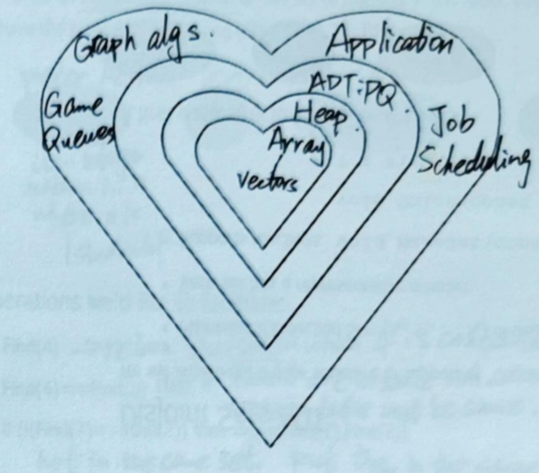
```
T heap<t>::removeMin(){
        T minVal = items[1];          grab the min.
        items[1] = items[size];       swap last element to front.
        size--;
        heapifyDown(1);.              restore heap property.
        return minVal;
}
```

```
void heap<t>::heapifyDown(int cIndex){
   if (hasAChild(cIndex)){ // 2*cIndex ≤ size.
      minChildIndex = minChild(cIndex); // deals with one child case.
      if (items[cIndex] > items[minChildIndex]{
         swap(items[cIndex], items[minChildIndex]).
         heapifyDown(minChildIndex).
      }
   }          O(log n).
}
```

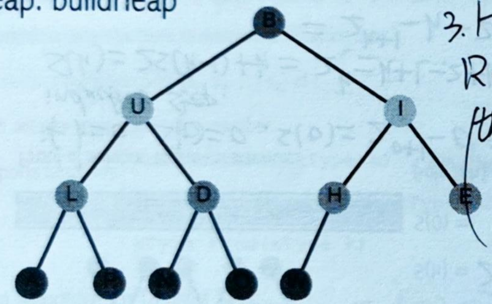
Announcements: PA3 due 03/30, 11:59p.

This image reminds us of a Heap., which is one way we can implement ADT Priority Queue, whose functions include insert and remove min/max. with running times $\Theta(\log n)$.

What have we done? Abstraction layers.



(min)Heap: buildHeap



3. Heapify Down from tail

Running time. $O(n)$

(the root is the only one that needs to go all the way down to the leaf others don't have to)

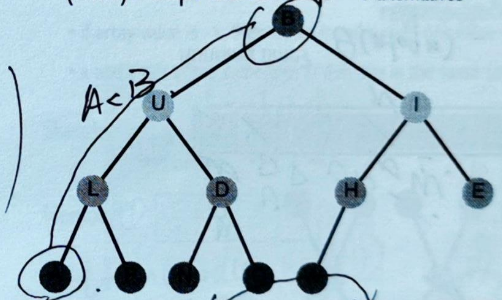
0												
~~Ø~~	B	U	I	L	D	H	E					n.

firstcell should be empty

1. Heapify Up from End: Not good. from beginning: Repeated insert. $\Theta(n\log n)$.

2. Sort the data: will satisfy heap property if the array is sorted, then any subsequence (trip from root to leaf) is also sorted. $\Theta(n\log n)$

(min)Heap: buildHeap - 3 alternatives



1. Sort the array: $\Theta(n\log n)$ merge sort.

B	U	I	L	D	H	E					

2. heapifyDown heapifyDown

```
void heap<t>::buildHeap(){
    for (int i=2;i<=size;i++)
        heapifyUp(i);
}
```

$\Theta(n\log n)$

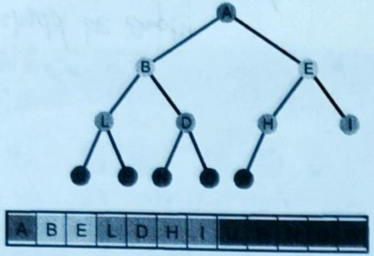
3. $\Theta(n)$.

```
void heap<t>::buildHeap(){
    for(int i=parent(size);i>0;i--)
        heapifyDown(i);
}
```

$\frac{n}{2}$ calls avoided by starting with parent.

Leaves are already heaps, no need to call heapifyDown on leaves.

(min)Heap: buildHeap



Thm: The running time of buildHeap on an array of size n is $\underline{\Theta(n)}$. $O(n)$ then argue $\Omega(n)$.

Instead of focusing specifically on running time, we observe that the time is proportional to the sum of the heights of all of the nodes, which we denote by S(h).

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

S(0) = 0.

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

Proof of solution to the recurrence:

if $h=0$ $S(0)=0$. $S(0)=2^{0+1}-0-2=0$ ✓.

inductive step.

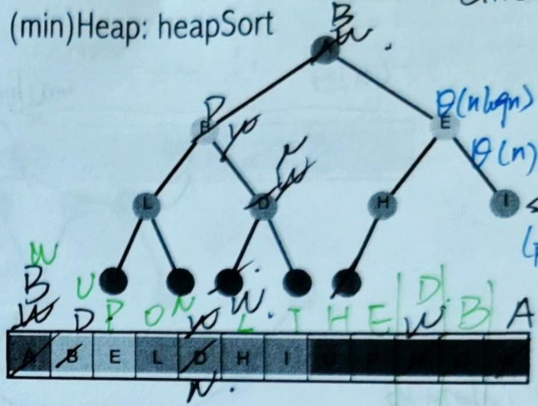
$$S(h)=2S(h-1)+h=2(2^{h}-(h-1)-2)+h$$
$$=2^{h+1}-h-2.$$

But running times are reported in terms of n, the number of nodes...

$T(n)=S(h)=2^{h+1}-h-2$, but we know $h\le \log_2 n$

$$\le 2^{\log_2 n+1}-\log_2 n-2. \le 2n-\log_2 n-2. \le 2n=\Theta(n).$$

(min)Heap: heapSort



Running time? $\Theta(n\log n)$.

Why do we need another sorting algorithm?

Same as mergeSort.

Given an unsorted array

1 build a heap. $\Theta(n)$.

$\Theta(n\log n)$ 2 Remove min n times placing min in the last spot position → size.

$\Theta(n)$ 3 Adjust the sort to satisfy expectation of caller. (Reverse array, place data at 0)

Depends on the "semi-sort" that the heap gives us, Otherwise, it's much like selection sort.

in-place algorithm.

not stable — relative locations of tied values are not maintained.

Announcements

PA3 available, due 03/30, 11:59p.

If this structure were a heap, we could build it in time $\Theta(n)$, and use it to support heapSort, which runs in time $O(n\log n)$. $\Omega(n\log n)$ lower bound in sorting. algorithm.

Disjoint Sets ADT all intersections of two sets will be empty (no intersection operation)

We will implement a data structure in support of "Disjoint Sets":

- Maintains a collection $S = \{s_0, s_1, \ldots s_k\}$ of disjoint sets.
- Each set has a representative member.
- Supports functions:

```
void MakeSet(kType k)
void Union(kType k1, kType k2)
kType Find(kType k)
```

constructor.

A first data structure for Disjoint Sets:

Find(7)=2.

0 1 4   2 7   3 5 6

representative.

array:

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

Find: returns arr[k] $\to \Theta(1)$

Union: Find(2) $\cup$ Find(3) $\to \Theta(n)$.

A Disjoint Sets example:

partitions a set into non-overlapping subsets.

Let R be an equivalence relation on the set of students in this room, where $(s,t) \in R$ if s and t have the same favorite among {FN, FB, TR, CC, PMC, AC}.

prefer AC.   CC   TR   FN ← convenience.

2 5 9   7   0 1 4 8   3 6 ← Disjoint Set.

ID - integer within 1 ... n where n is |students|

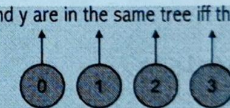
Operations we'd like to facilitate:

1. Find(4) → return the $\boxed{\text{set}}$ in which 4 is contained (returns same $\boxed{\text{"representative"}}$ of the set, 0, 1, 4, 8, any is fine)
2. Find(4)==Find(8) True if 4 and 8 are in same set. representative must be same.
3. If (!(Find(7)==Find(2)) then Union(Find(7),Find(2))
   not in the same set. put them in the same set by taking union.
   After union, all elements of the resulting set have the same representative.

A better data structure for Disjoint Sets: UpTrees

- if array value is -1, then we've found a root (representative), o/w value is index of parent which is in the same set as current index.
- x and y are in the same tree iff they are in the same set.

Sketch:



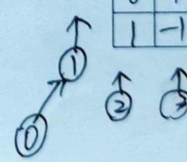
Memory: (1)

0	1	2	3
-1	-1	-1	-1

Union(Find(0), Find(1))
↳0 ↳1

(2)

0	1	2	3
1	-1	-1	-1



Union(Find(2), Find(0)).
2. 1. walk up the tree.

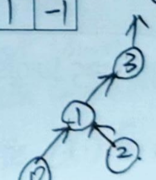
(3)

0	1	2	3
1	-1	1	-1

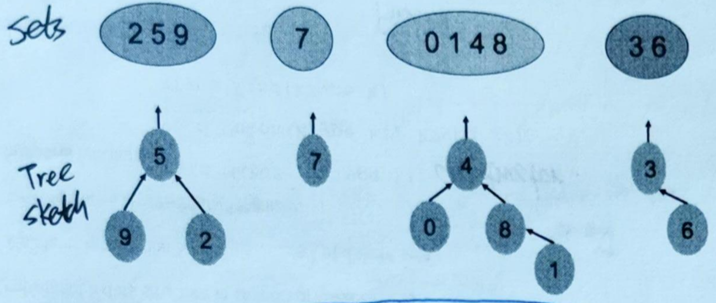
(4)

0	1	2	3
1	3	1	-1

Union(Find(0), Find(3))
2.



Example: partition of elements into disjoint sets



array

0	1	2	3	4	5	6	7	8	9
4	8	5	~~6~~ -1	-1	-1	~~-1~~ 3	-1	4	5

Find(1)

UpTree implementation of Disjoint Sets ADT:

```
int DS::Find(int i) {
        if (s[i] < 0) return i;
        else return Find(s[i]);
}
```

Running time depends on height of the tree.

Worst case? $\Theta(n)$.

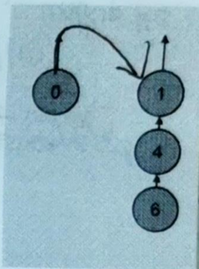
← too expensive

What's an ideal tree?

constant height.

```
int DS::Union(int root1, int root2) {
        s[root2] = root1;
}
```

Running time $\Theta(1)$.

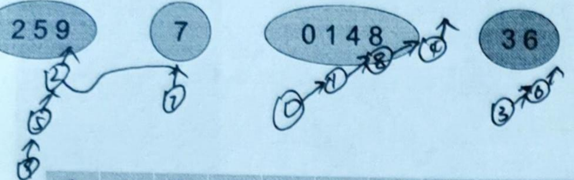


...nouncements

...available, due 03/30, 11:59p. GS submission this evening.

...call, *Disjoint Sets* example:

...se the members of each set below have the same favorite among {AC, FN, FB, TR, CC, CR}.



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

uptrees in an array.

...are the results of the following statements?

...(8) = 4. (return the set in which 8 dwells) ADT – Find, Union.

...(0)==Find(8) True

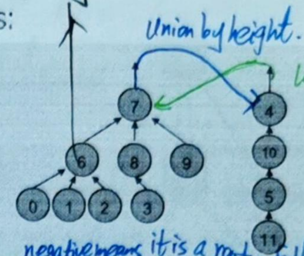
...Find(7)==Find(9)) then Union(Find(7),Find(9)) pointing 7 to 2 keeps tree short.

Smart unions:

array indices

Union by height.

union by size. find time increased



negative means it is a root, followed by its height.

Union by height: store −(height+1) in root value.

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

Keeps overall height of tree as small as possible.

Union by size: store −(# of nodes) in root value.

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

7 -12.

Increases distance to root for fewest nodes.

Both of these schemes for Union guarantee the height of the tree is $\Theta(\log n)$.

...art unions:

```
int DS::Find(int i) {
     if (s[i] < 0) return i;
     else return s[i]= Find(s[i]);
}
```

assign the parent and return the value.

Keeps the same. $\Theta(\log n)$. improved from $\Theta(n)$.

```
int DS::UnionBySize(int root1, int root2) {
     int newSize = s[root1] + s[root2];
     if (isBigger(root1,root2)){
          s[root2] = root1;
          s[root1] = newSize;
     }
     else {
          s[root1] = root2;
          s[root2] = newSize;
     }
}
```

↳ store negative magnitudes.

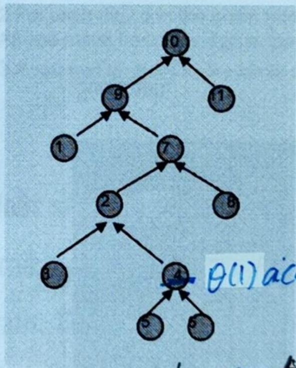
resulting root.

by making union more complex, we reduce running time of find.

Runtime $\Theta(1)$.

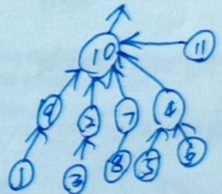
Path Compression:

Find(4).



$\Theta(1)$ access from array.

example execution.



along the path from a node to the root, it redirects all nodes on the path to the root.

Uptrees
Smart Union
Path Compression
We hope $\Theta(1)$ for all x.

Analysis:

Something we'll need -- Iterated log $\log^* n := \begin{cases} 0 & \text{if } n \le 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$

counts the number of time we can take log of a number before we hit 1.

Example:

$2^{65536} \xrightarrow{1} 65536 \xrightarrow{2} 16 \xrightarrow{3} 4 \xrightarrow{4} 2 \xrightarrow{5} 1$. $\log^* 2^{65536} = 5$.

Relevant result:

In an upTree implementation of Disjoint Sets using smart `union` and `find` with path compression...

large number.

any sequence of m `union` and `find` operations results in worst case running time of $O(m \log^* n)$, where n is the number of items.

nearly constant.

actually a loose bound.

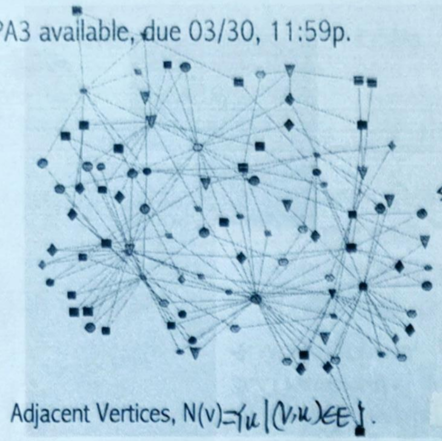
http://research.cs.vt.edu/AVresearch/UF/

...nouncements PA3 available, due 03/30, 11:59p.

...phs!! G = (V,E)

2003 internet.

...ph: vertex: router
edge: network



subgraph: G' = (V', E') is a subgraph of a graph G iff $V' \subseteq V$, $E' \subseteq E$, $e = (v,u) \in E'$, then $v \in V'$, $u \in V'$.

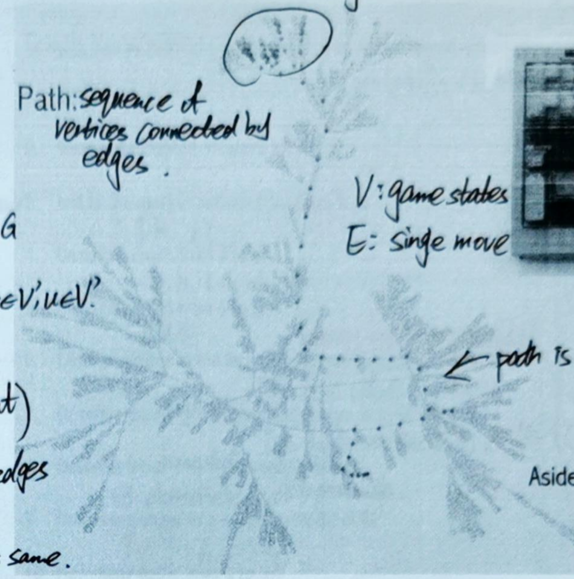
degree of vertice: (# of neighbors that v has)

Adjacent Vertices, $N(v) = \{u \mid (v,u) \in E\}$.

Incident Edges, $I(v) = \{(u,v) \mid (u,v) \in E\}$. or $I(v)$ # of incided edges

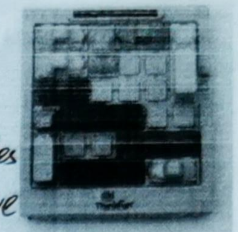
{(■,▼), (■,◆)}.

Aside: Vertices are labelled, no pair of adjacent vertices is the same.



winning state

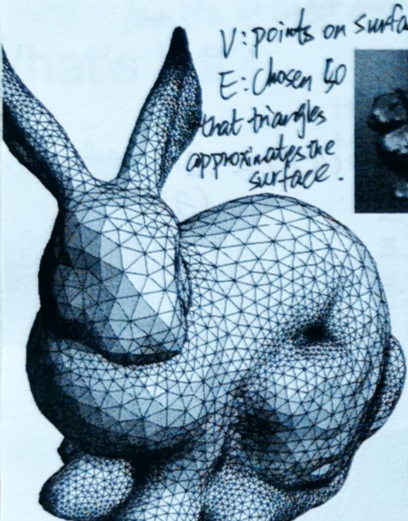
Path: sequence of vertices connected by edges.



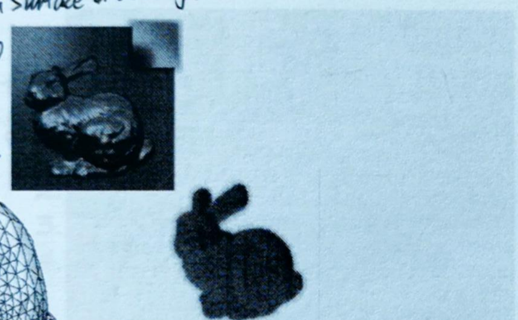
V: game states
E: singe move

path is the solution to the puzzle

Aside: there is a great alg for solving this one — BFS.



V: points on surface of an object.
E: chosen so that triangles approximates the surface.

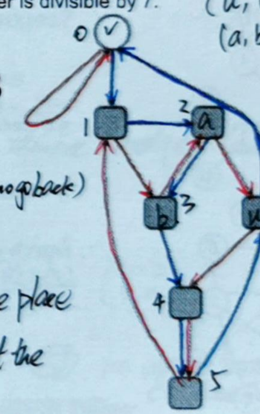


Planar surface mesh.
– Can be embedded in a plane drawn with no crossing edges.

This graph is used to calculate whether a given number is divisible by 7.

1. Start at the circle node at the top.
2. For each digit d in the given number, follow d blue (solid) edges in succession. As you move from one digit to the next, follow 1 red (dashed) edge.
3. If you end up back at the circle node, your number is divisible by 7.

3703



directed graph: $(u,v) \neq (v,u)$. $(a,b) \neq (b,a)$.

V: k is k mod 7.

Walk: sequence of vertices between which there are edges.

Trail: walk with no repeated edges. (can no go back)

simple Path: trail with no repeated vertices

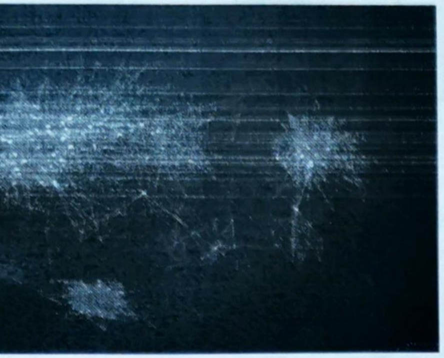
Circuit: trail that begins and ends at same place

Cycle: path that begins and ends at the same place. (path that allows one repeat).

not a path
but it is a trail

nouncements PA3 available, due 03/30, 11:59p.

phs!! G = (V,E)



Tree: connected, acyclic graph.
≡ between every pair of vertices, there is one path.

Simple (sub)Graph:
Graph with no self loops
multiedges.
(u,v) ~~(u,x)~~ (u,v) is unique ~~u~~ v

Spanning subgraph:
Let G' be a subgraph of G.
$G' = (V', E')$, $G = (V, E)$
G' is a spanning subgraph if $V' = V$

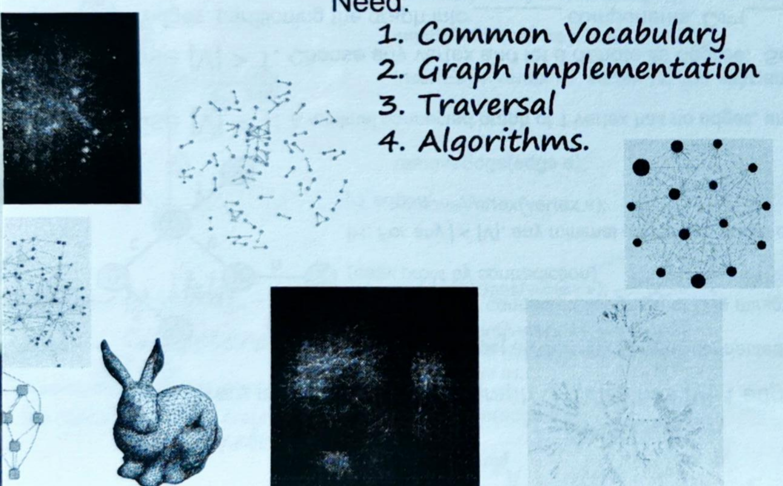
nected (sub)Graph: A graph is connected if between any pair of vertices connected.

Spanning tree:
G' is a spanning tree of G if it is a tree and $V' = V$.

nected component: there is a path.
maximal connected subgraph.
if we add anything to subgraph, then it is no longer connected.

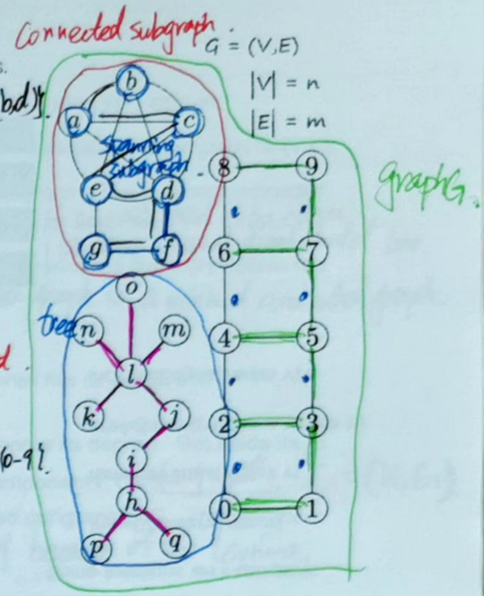
What's left?

How do we get from here to there?
Need:
1. Common Vocabulary
2. Graph implementation
3. Traversal
4. Algorithms.



Graph Vocabulary: Use the graph G to answer these questions.

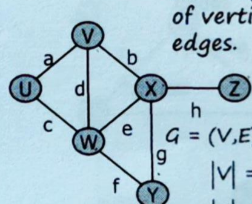
1. List the edges incident on vertex b: {(b,a), (b,c), (b,e), (b,d)}.
2. What is the degree of vertex h? 3.
3. List all the vertices adjacent to vertex i: {h, j}.
4. Describe a path from p to o: p, h, i, j, l, o
5. Describe a path from 6 to g: DNE.
6. List the vertices in the largest complete subgraph in G: every node is connected. {a, b, c, d, e} = K_5
7. Describe the largest connected subgraph in G: Tie between the [p, o] and [0-9].
8. Describe the connected components in G: 3 connected components.
9. How many edges in a spanning forest of G? 9+9+6=24.
10. How many paths from 0 to 9? $2^4 = 16$ (encoding of binary #)
11. Can you draw G with no crossing edges? not possible for red component. Cannot draw K5 in the plane.
if a graph has a K_5 minor, then it is not planar.



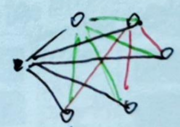
Connected subgraph is not necessarily a connected component.
e.g. ④–⑤ is connected subgraph, but not a connected component.

Graphs: theory that will help us in analysis

Running times often reported in terms of n, the number of vertices, but they often depend on m, the number of edges.



G = (V,E)
|V| = n
|E| = m



n-1
+ n-2
+ n-3
+
⋮
+ 1
$= \frac{n(n-1)}{2}$

n vertices, each with (n-1) incident edges.
n(n-1) end points.
$\Rightarrow \frac{n(n-1)}{2}$ edges.

How many edges?

At least:

connected – n-1
we can remove an edge in a cycle to make it a tree.

not connected – 0

At most:

Complete graph.
simple – $\frac{n(n-1)}{2}$. (C_n^2).

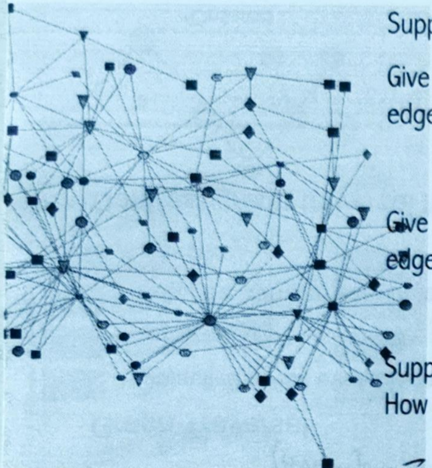
not simple – no upper bound
can have multiple edges, self loops.

Relationship to degree sum:

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

nouncements PA3 available, due 04/09, 11:59p.

phs!! G = (V,E)



Suppose n = 100,

Give a lower bd for the number of edges: 99

$|E| = |V| - 1$ for a minimally connected graph (tree).

Give an upper bd for the number of edges:

$m \le 4950 = \frac{n(n-1)}{2}$

Suppose the average degree is 22. How many edges?

# of incident edges of a vertex.

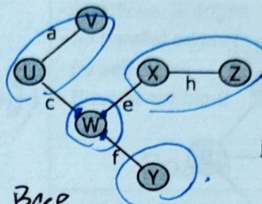
$$\frac{\sum_{v \in V} deg(v)}{n} = 22$$

$$\Rightarrow \sum_{v \in V} deg(v) = 2200 \Rightarrow 1100 \text{ edges.}$$

every edge is counted twice.

$$2|E| = \sum_{v \in V} deg(v)$$

Thm: Every minimal connected graph G=(V,E) has |V|-1 edges.



Proof: Consider an arbitrary minimal connected graph G=(V,E).

Lemma: Every connected subgraph of G is minimally connected. (easy proof by contradiction) if subgraph had extra edges, then the original graph would too.

IH: For any j < |V|, any minimal connected graph of j vertices has j-1 edges. any smaller graph is a minimal connected graph.

Base

Suppose |V| = 1: A minimal connected graph of 1 vertex has no edges, and 0 = 1-1.

Inductive.

Suppose |V| > 1: Choose any vertex w and let d denote its degree. Set aside its d incident edges, partitioning the graph into $d+1$ components, C_0=($\{w\}$, $\emptyset$), ... $C_1 = (V_1, E_1)$, C_d=(V_d, E_d), each of which is a minimal connected subgraph of G. This means that $|E_k|$ = $|V_k| - 1$ by IH. which we can apply because of the lemma, by observing that C_k is smaller than G.

Now we'll just add up edges in the original graph:

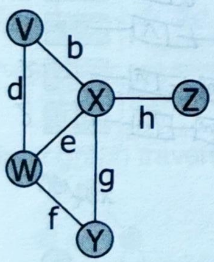
$$|E| = d + |E_1| + |E_2| + \cdots + |E_d|$$

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

$$= \sum_{k=1}^{d} |V_k| = |V| - 1 = n - 1$$

all vertices except w.

raphs: Toward implementation…(ADT)



Functions: (merely a smattering…)

insertVertex(pair keyData)

insertEdge(vertex v1, vertex v2, pair keyData)

removeEdge(edge e);

removeVertex(vertex v);

incidentEdges(vertex v); → set of neighbors

areAdjacent(vertex v1, vertex v2); existence of an edge.

origin(edge e);

destination(edge e);

$u \xrightarrow{e} v$ origin(e) → u dest(e) → v $(u,v) \neq (v,u)$

ta:

Vertices — hashtable.

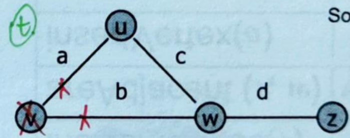
Edges — includes endpoint info hashtable

\+ some structure that reflects the connectivity of the graph

2 alternatives.

Graphs: Adjacency Matrix

$G = (V, E)$ $|V| = n$ $|E| = m$



Some functions we'll compare:

insertVertex(vertex v) naive implementation is $\Theta(n^2)$ per addition. $\Theta(n)$ if we double and copy.

removeVertex(vertex v) $\Theta(n)$ to remove all incident edges.

areAdjacent(vertex v, vertex u) $\Theta(1)$ justifies choice of matrix

incidentEdges(vertex v) $\Theta(n)$ to traverse a row of the matrix

Vertex $\Theta(1)$ amortized (resize). $\Theta(1)$ amortized

Vertex
u
~~v~~
w
z

edge.

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

n×n matrix.

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

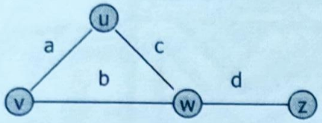
Could just be booleans.

Symmetric matrix. "need" to store only half. $\frac{n^2}{2}$ asymptotically still n^2.

at cost n^2, we get n future additions $\Rightarrow \Theta(n)$ amortized.

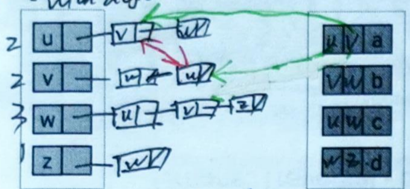
Announcements PA3/HW3 av, due 04/09, 11:59p.

Graphs: Adjacency List



vertex
augment vertex info with degree

edge



2m total nodes

storage is $\Theta(n+m) = O(n^2)$
(max of n+m)

Some functions we'll compare:

insertVertex(vertex v) $\Theta(1)$ amortised.

removeVertex(vertex v) $\Theta(deg(V))$

areAdjacent(vertex v, vertex u) $\Theta(\min\{deg(v), deg(u)\})$

incidentEdges(vertex v)
$\Theta(deg(v))$

$m = O(n^2)$.
or $\Theta(n)$
$\Theta(1)$
} Anything $\leq n^2$.

Graphs: Asymptotic Performance

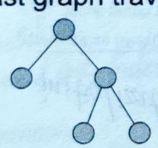
• n vertices, m edges • no parallel edges • no self-loops } simple • Bounds are big-Θ	Edge List if no additional structure.	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	~~n^2~~ $\Theta(n)$ amortized
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	~~n^2~~ $\Theta(n)$ amortized.
removeEdge(e)	1	1	1

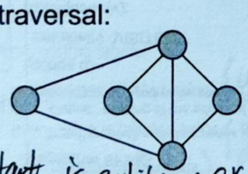
Graph Traversal

Objective: Visit every vertex and every edge in the graph,
while honoring the connectivity of the graph, walk around on edges of graph.

Purpose: We can search for interesting substructures in the graph,
like shortest path from current location to another vertex.
list of low degree

Contrast graph traversal to BST traversal:





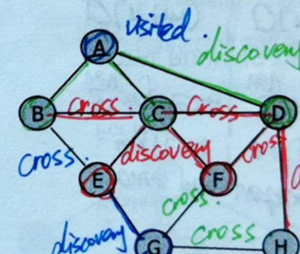
- Ordered
- Obvious start

good way of knowing when done

- start is arbitrary or application dependent.
- iterate over incident edges
- mark progress

Graph Traversal: Breadth First Search (by example)

p - predecesor
d - distance



vertex structure / adj list data.

p	d		
-	0	A	C B D
A	1	B	A C E
A	1	C	B A D E F
A	1	D	A C F H
C	2	E	B C G
C	2.	F	C D G
E.	3	G	E F H
D	2	H	D G

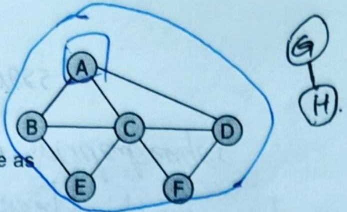
Queue.
~~G H F E D B C A~~
0 1 1 1 1 2 2
A C B D E F H G.

For every neighbor of current vertex.
1. mark visited.
2. edge between is discovery.
3. update p & d, d = 1 + current d.
p = current.
4. put on queue.

① Discovery edges are a spanning tree
② d gives the length of shortest path from start. to every other vertex.
③ p gives the paths (like up tree).

Graphs: Traversal – BFS

Visits every vertex and classifies each edge as either “discovery” or “cross”



initialization

Algorithm BFS(G)

Input: graph G

Output: labeling of the edges of G as discovery edges and back edges

```
For all u in G.vertices()
  setLabel(u, UNEXPLORED)
For all e in G.edges()
  setLabel(e, UNEXPLORED)
For all v in G.vertices()
  if getLabel(v) = UNEXPLORED
    BFS(G,v)
```

grabs all components of graph.

Algorithm BFS(G,v)

Input: graph G and start vertex v

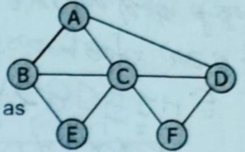
Output: labeling of the edges of G in the connected component of v as discovery edges and cross edges

```
queue q;
setLabel(v, VISITED)
q.enqueue(v);
While !(q.isEmpty)
  q.dequeue(v)
  For all w in G.adjacentVertices(v)
    if getLabel(w) = UNEXPLORED
      setLabel((v,w),DISCOVERY)  } update p/d.
      setLabel(w, VISITED)
      q.enqueue(w)
    else if getLabel((v,w)) = UNEXPLORED
      setLabel((v,w),CROSS)
```

pseudo code for the algorithm.

Graphs: Traversal – BFS

Visits every vertex and classifies each edge as either “discovery” or “cross” $\Omega(n+m)$



While loop

once per vetex.

adj List	adj matrix

For loop

$\sum_{v \in V} \deg(v) = 2m$	$\sum_{v \in V} n = n^2.$

TOTAL RUNNING TIME:

$O(n+m)$	$O(n^2)$

General observations:

- running time depends on implementation
- BFS gives shortest path with p/d book keeping
- easy to count components & detect cycles

Algorithm BFS(G,v)

Input: graph G and start vertex v

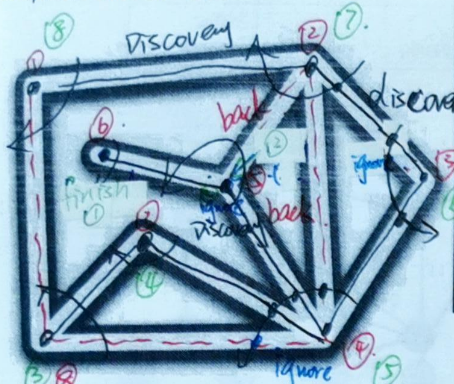
Output: labeling of the edges of G in the connected component of v as discovery edges and cross edges

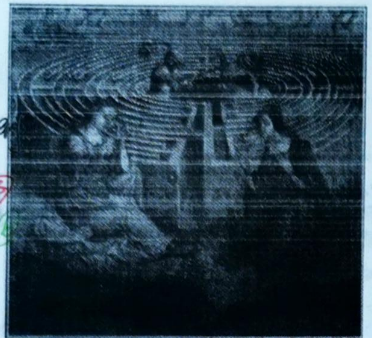
```
queue q;
setLabel(v, VISITED)  O(1)
q.enqueue(v);  O(1)
While !(q.isEmpty)
  q.dequeue(v)  O(1)
  For all w in G.adjacentVertices(v)  deg(v)
    if getLabel(w) = UNEXPLORED
      setLabel((v,w),DISCOVERY)  } O(1)
      setLabel(w, VISITED)
      q.enqueue(w)
    else if getLabel((v,w)) = UNEXPLORED
      setLabel((v,w),CROSS)
```

BFS runtime $\Theta(n+m)$.

Announcements PA3/HW3 av, due 04/09, 11:59p.

Graphs: Traversal - DFS





DFS(v)

Ariadne, Theseus, and the Minotaur

1. visits a vertex v.
2. for each neighbor u
   if u unvisited.
   marks (u,v) as discovery
   DFS(u).
   else marks (u,v) as back.

http://www.cs.duke.edu/csed/jawaa2/examples/DFS.html

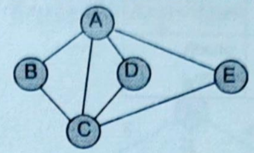
http://www.student.seas.gwu.edu/~idsv/idsv.html

http://www.youtube.com/watch?v=8qrZ1clEp-Y

in graph: Discovery
Back
Ignore.
Finish.

Discovery edges
spanning tree.
not shortest path.

DFS: "visits" each vertex, classifies each edge as either "discovery" or "back"



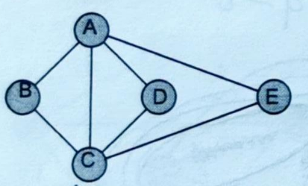
```
Algorithm DFS(G)
    Input: graph G
    Output: labeling of the edges of G
    as discovery edges and back edges

For all u in G.vertices()
    setLabel(u, UNVISITED)
For all e in G.edges()
    setLabel(e, UNEXPLORED)
For all v in G.vertices()
    if getLabel(v) = UNVISITED
        DFS(G,v)
```

} Initialization.

} picks up components.

```
Algorithm DFS(G,v)
    Input: graph G and start vertex v
    Output: labeling of the edges of G in the
    connected component of v as discovery edges and
    back edges

setLabel(v, VISITED)
For all w in G.adjacentVertices(v)
    if getLabel(w) = UNVISITED
        setLabel((v,w),DISCOVERY)
        DFS(G,w)
    else if getLabel((v,w)) = UNEXPLORED
        setLabel(e,BACK)
```

Graphs: DFS example



A	B C D E
B	A C
C	B A D E
D	A C
E	A C

Application:
Topological sort, Eulerian paths.

General observations:

- Discovery edges form spanning tree.
- 2 convenient visit times.
- can Count componenets.
  detect edges.

setting/getting labels:

every vertex labeled twice { unexplored. visited.

$\Omega(n+m)$.

every edge is labeled twice { unexplored. discovery/back.

querying vertices: (for loop)

each vertex $\deg(v)$ (adj list) | adj matrix. n n^2.

total over algorithm $\sum_{v\in V} \deg(v) = 2m$.

querying edges: $\Theta(1)$ per edge.

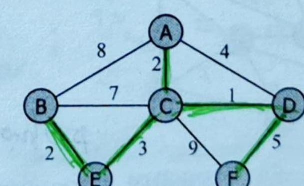
TOTAL RUNNING TIME:

$\Theta(n+m)$ adjacency list.

$\Theta(n+n^2)=\Theta(n^2)$ adj matrix.

Minimum Spanning Tree Algorithms:

- Input: connected, undirected graph G with unconstrained edge weights positive/negative.
- Output: a graph G' with the following characteristics -
  - G' is a spanning subgraph of G
  - G' is connected and acyclic (a tree)
  - G' has minimal total weight among all such spanning trees -



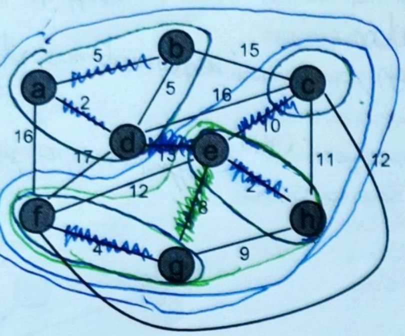
① changing BC 7 to 4 does not change the MST.

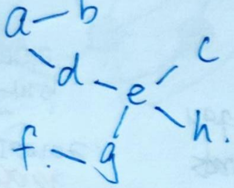
② change CF 9 to 4 will change the MST (DF to CF).

Observation:
if we partition the vertices into 2 sets, and look at the edges between, then least weight edge is part of some MST.
We don't care about total path length between any specific pair.

edges in increasing cost.

Kruskal's Algorithm





priority queue (m entries).

↓ increasing cost.

~~(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)

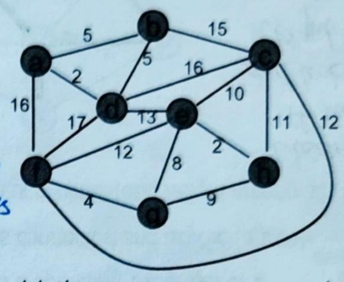
reject (makes a cycle)

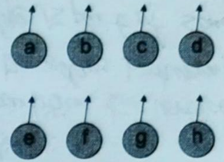
(g,h) g & h are in the same component. (g,h) creates a cycle.

maintain the growth of the structure as a collection of disjoint set.

we are done because we have added n-1 edges.

Kruskal's Algorithm (1956)





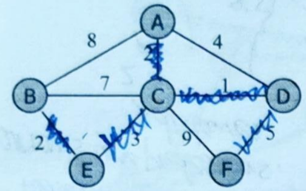
(a,d)
(e,h)
(f,g)
(a,b)
(b,d)
(g,e)
(g,h)
(e,c)
(c,h)
(e,f)
(f,c)
(d,e)
(b,c)
(c,d)
(a,f)
(d,f)

1. Initialize graph T whose purpose is to be our output. Let it consist of all n vertices and no edges.

2. Initialize a disjoint sets structure where each vertex is represented by a set.

3. RemoveMin from PQ. If that edge connects 2 vertices from different sets, add the edge to T and take Union of the vertices' two sets, otherwise do nothing. Repeat until n-1 edges are added to T.

Announcements PA3/HW3 av, due 04/09, 11:59p.

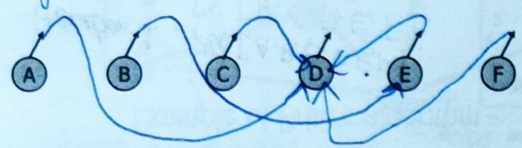


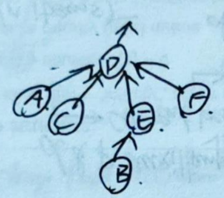
~~(c,d)~~
~~(e,b)~~
~~(e,a)~~
~~(e,c)~~
~~(a,d)~~
~~(d,f)~~
(b,c)
(b,a)
(c,f)

m edges.

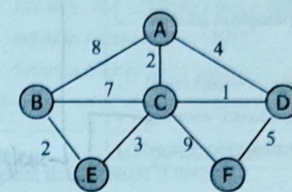
(n-1)
5 edges ⇒ tree done

Disjoint sets.





Kruskal's Algorithm - preanalysis



Algorithm *KruskalMST(G)*

disjointSets forest;
for each vertex *v* in *V* do
forest.makeSet(v);

PQ { *priorityQueue Q;*
Insert edges into *Q*, keyed by weights

graph *T* = (*V*,*E*) with *E* = ∅; $\Theta(n)$.
we know we only need $\Theta(n)$ edges.
while *T* has fewer than *n*-1 edges do
edge *e* = *Q.removeMin()*
Let *u*, *v* be the endpoints of *e*
if *forest.find(v)* ≠ *forest.find(u)* then
$\Theta(1)$ → Add edge *e* to *E* $O(1)$ assume.
forest.smartUnion
(forest.find(v),forest.find(u))

return *T*

sorting algorithm $\Theta(m\log n)$.

Priority Queue:	Heap	Sorted Array
To build	$\Theta(m)$.	$\Theta(m\log n)$
Each removeMin	$\Theta(\log n)$	$\Theta(1)$.

we are building an array out of m items, also
$\Theta(\log m)=\Theta(\log n)$.

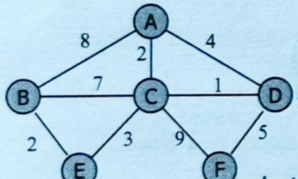
pops top
restoreheap (heapify down)

$m \le n^2$.
$\log m \le \log n^2$
$\log m \le 2\log n$.
$\Theta(\log n)=\Theta(\log m)$.

add Edge -
① AdjList add edge to start of each of two lists. $O(1)$.

Kruskal's Algorithm - analysis

$\Theta(m\log n)$



m iterations

no iterations over incident edges
⇒ graph implementation doesn't matter.

Algorithm *KruskalMST(G)*

disjointSets forest;
for each vertex *v* in *V* do
forest.makeSet(v); $\Theta(n)$.

priorityQueue Q; $\Theta(m)$ vs $\Theta(m\log n)$.
Insert edges into *Q*, keyed by weights

graph *T* = (*V*,*E*) with *E* = ∅; $\Theta(n)$.

while *T* has fewer than *n*-1 edges do
edge *e* = *Q.removeMin()* $\Theta(\log n)$ vs $\Theta(1)$.
Let *u*, *v* be the endpoints of *e*
if *forest.find(v)* ≠ *forest.find(u)* then
Add edge *e* to *E*
forest.smartUnion
(forest.find(v),forest.find(u)) } $\Theta(1)$.

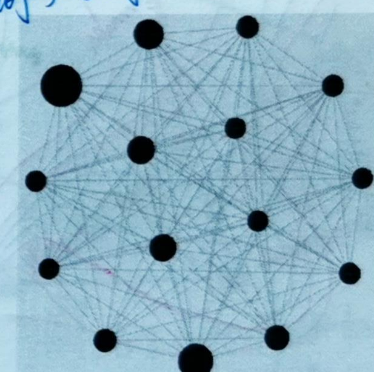
return *T*

Graph Impl -
answer - adjlist.
but it doesn't matter.

Priority Queue:	Total Running time:
Heap	$\Theta(n+m+n+m\log n+m)=\Theta(n+m+m\log n)=\Theta(m+m\log n)$
Sorted Array	$\Theta(n+m\log n+n+m)=\Theta(n+m+m\log n)=\Theta(m+m\log n)$.

} tie.

we may choose sorted array, since the graph may have already been sorted.
→ we can make improvement on $(m\log n)$



→ adj matrix.

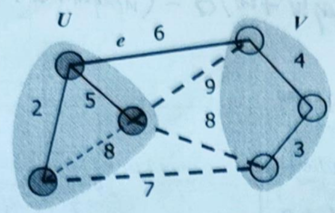
How would you characterize the difference between these two graphs?

connectedness.
completeness.

Dense graph: right graph has many edges $\Theta(n^2)$
Sparse graph: left has few edges. $\Theta(n)$

adjacency lists.

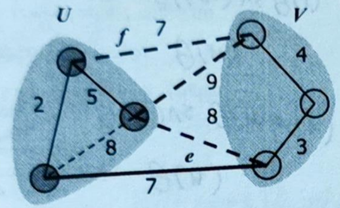
Prim's algorithms (1957) is based on the Partition Property:



Consider a partition of the vertices of G into subsets U and V.

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

Then e is part of some minimum spanning tree.

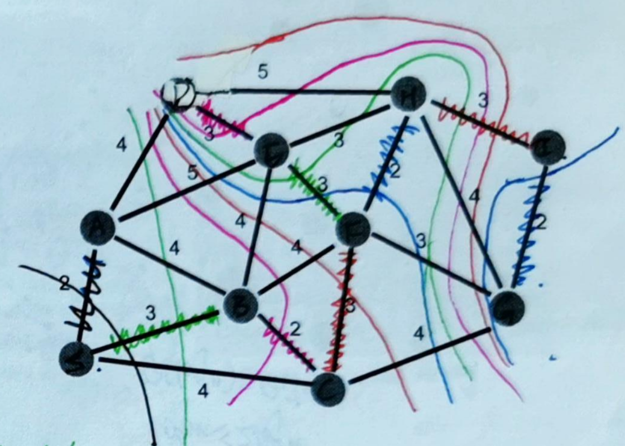


Proof:

See cpsc320?

MST - minimum total weight spanning tree

Theorem suggests an algorithm... (like BFS, with priority queue)

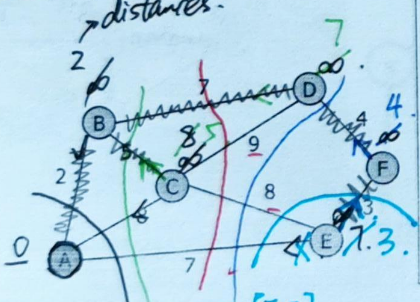


Solved / Done "labelled"

d-distance (cost).
p- predecessor

Example of Prim's algorithm -

iteration 1. PQ [A,B,C,D,E,F].
iteration 2 PQ [B,C,D,E,F].
iteration 3 PQ [C,D,E,F].
iteration 4 PQ [D,E,F].
arrows are predecessors.
→ distances.



iteration 5 PQ [E,F]

d & p auxilliary structure

Initialize structure: ← current best cost of attaching to solution
1. For all v, d[v] = "infinity", p[v] = null ↳ (u,v) is the edge that gives best.
2. Initialize source: d[s] = 0
3. Initialize priority (min) queue of d[v] (n items)
4. Initialize set of labeled vertices to Ø. gives partition.

Repeat these steps n times:
- Find & remove minimum d[] unlabelled vertex: v PQ operation.
- Label vertex v

BFS.

- For all unlabelled neighbors w of v,
  If cost(v,w) < d[w]
  d[w] = cost(v,w) } update.
  p[w] = v

Final solution is in the predecessors.
(no unique solution).

Prim's Algorithm (undirected graph with unconstrained edge weights):

Initialize structure:
1. For all v, d[v] = "infinity", p[v] = null
2. Initialize source: d[s] = 0
3. Initialize priority (min) queue
4. Initialize set of labeled vertices to Ø.

PQ operations

Repeat these steps n times:
- Remove minimum d[] unlabeled vertex: v
- Label vertex v (set a flag)
- For all unlabeled neighbors w of v,
  If cost(v,w) < d[w]
  d[w] = cost(v,w) changes a priority.
  p[w] = v

	adj mtx	adj list
heap	O(n^2 + m log n)	O(n log n + m log n)
Unsorted array	O(n^2)	O(n^2)

Which is best?

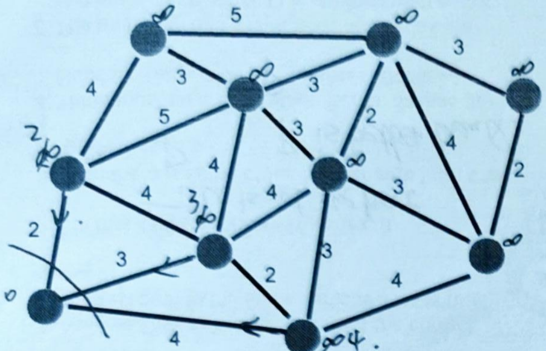
Depends on density of the graph:

Sparse

Dense

Announcements PA3/HW3 av, due 04/09, 11:59p.

Prim's MST : weighted, undirected edges



For each vertex.
we may update d/p
deg(v) times

- choose an arbitrary start. —choice may change final tree but will not change final value of MST (value == total cost)
- init structure p/d structures
- proceed algorithm.

Prim's Algorithm (undirected graph with unconstrained edge weights):

Initialize structure:

1. For all v, d[v] = "infinity", p[v] = null $\Theta(n)$.
2. Initialize source: d[s] = 0 $\Theta(1)$.
3. Initialize priority (min) queue $\Theta(n)$ $\Theta(1)$
4. Initialize set of labeled vertices to ∅. $\Theta(1)$

Repeat these steps n times: ← # of vertices.

- Remove minimum d[] unlabeled vertex: v n log n n^2 total
- Label vertex v (set a flag) $\Theta(1)$

  $\sum_v \deg(v)\log n = m\log n$ total n^2
- For all unlabeled neighbors w of v (depend on graph impl)
  - If cost(v,w) < d[w] $\Theta(1)$
  - changes PQ. { d[w] = cost(v,w) log n. $\Theta(1)$ hash table.
    p[w] = v

PQ \ graph impl.	adj mtx	adj list
heap	~~O(n² + m log n)~~	n + n log n + m log n. O(n log n + $\boxed{\text{m log n}}$)
Unsorted array	O(n²)	O(n²) $O(n^2+m)$.

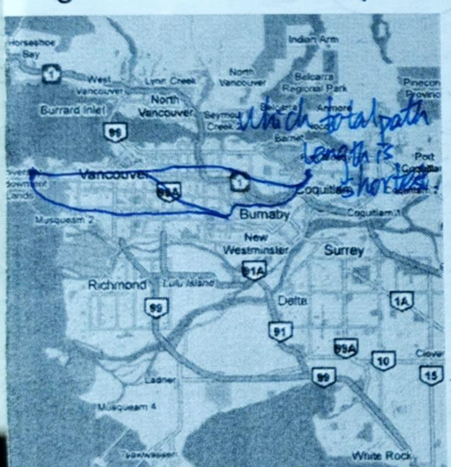
Which is best?

Depends on density of the graph:

Sparse (if m is $O(n)$) : heap + adj list.

Dense unsorted array.

Single source shortest path

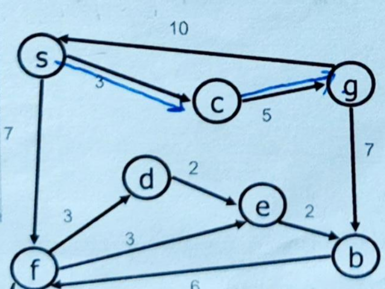


- if a graph has a neg weight cycle, then shortest path doesn't exist.
- if a graph has neg-weight edge and no neg weight cycle, then shortest paths exist, but Dijkstra's won't find them

Given a start vertex (source) s, find the path of least total cost from s to every vertex in the graph. ← dist, time, $

Input: directed graph G with non-negative edge weights, and a start vertex s.

Output: A subgraph G' consisting of the shortest (minimum total cost) paths from s to every other vertex in the graph. Dijkstra's Algorithm (1959)



Single source shortest path (directed graph w non-negative edge weights):

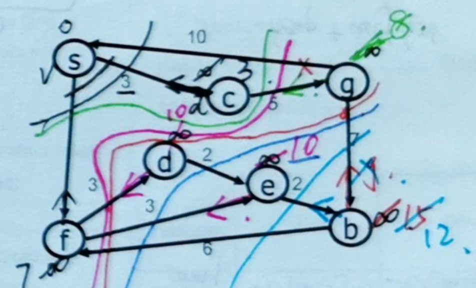
Initialize structure:

1. For all v, d[v] = "infinity", p[v] = null
2. Initialize source: d[s] = 0
3. Initialize priority (min) queue

Repeat these steps n times:

- Find minimum d[] unlabelled vertex: v

  s c f g
- Label vertex v
- For all unlabelled outgoing. neighbors w of v,
  - If (d[v] + cost(v,w) < d[w])
  - d[w] = d[v] + cost(v,w).
  - p[w] = v



d[v] contain path lengths
p[v] contain paths themselves in reverse order.

ADT is a description of the functionality of a data structure

Single source shortest path (directed graph w non-negative edge weights):

Initialize structure:

1. For all v, d[v] = "infinity", p[v] = null
2. Initialize source: d[s] = 0
3. Initialize priority (min) queue

Repeat these steps n times:

- Find minimum d[] unlabelled vertex: v
- Label vertex v
- For all unlabelled neighbors w of v,

  If d[v] + cost(v,w) < d[w]

  d[w] = d[v] + cost(v,w) same cost.

  p[w] = v

analysis is the same as prims.

~~Prim's~~ Dijkstra's Algorithm (undirected graph with unconstrained edge weights):

Initialize structure:

1. For all v, d[v] = "infinity", p[v] = null
2. Initialize source: d[s] = 0
3. Initialize priority (min) queue
4. Initialize set of labeled vertices to ø.

Repeat these steps n times:

- Remove minimum d[] unlabeled vertex: v
- Label vertex v (set a flag)
- For all unlabeled neighbors w of v,

  If d[v] + cost(v,w) < d[w]

  d[w] = d[v] + cost(v,w)

  p[w] = v

	adj mtx	adj list
heap / sorted array	$O(n^2 + m \log n)$ $= O(m \log n)$	$O(n \log n + m \log n)$ $= O(m \log n)$
Unsorted array	$O(n^2)$	$O(n^2)$

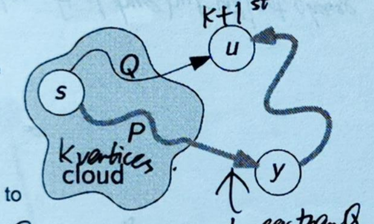
Which is best?

Depends on density of the graph:

Sparse heap + adj list.

Dense unsorted array.

Dijkstra's correctness:



1. Assume Dijkstra's algorithm finds the correct shortest path to the first k vertices it visits (the cloud). …
2. But that it fails on the k+1st vertex, u. — Q is not the best
3. Then there is some other, shorter path from s to u. Call it P. P is better than Q.
4. There must be a node other than u, outside the cloud through which P passes. Call it y.

   if d[y] < d[u], then algorithm would have chosen y instead of u.
5. The path from s to y is at least as long as the path from s to u, since Q is shortest path out of the cloud. if it weren't, we should have chosen y instead of u.
6. P is even longer!! But that's a contradiction. than Q.
7. So our assumption that we failed on the k+1st vertex is incorrect.

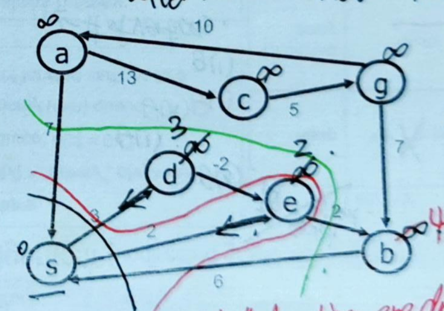
Let u be the last vertex (k+1=n).

Single source shortest path (directed graph w non-negative edge weights):

Dijkstra's Algorithm (1959)

Why non-negative edge weights??

neg edge weights may reduce total path length after we call a vertex finished.



Initialize structure:

Repeat these steps:

1. Label a new (unlabelled) vertex v, whose shortest distance has been found
2. Update v's neighbors with an improved distance

all labelled vertices are done. but s→d→e is shorter than se. and we do not improve it.