Data Structures

Review

CS 225

Brad Solomon

December 6, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Material covered here is not only material in class!

Represents only an attempt to provide some helpful resources.

Exam is comprehensive

↳ Multiple coding questions

- Graph question
-

↳ Multiple choice questions

Requested: Iterators

Tree → post order traversal
yes an iterator

```
1
2  std::vector<Animal> zoo;
3
4
5  /* Full text snippet */
6
7    for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
8      std::cout << (*it).name << " " << (*it).food << std::endl;
9    }
10
11
12 /* Auto Snippet */
13
14   for ( auto it = zoo.begin(); it != zoo.end; ++it ) {
15     std::cout << animal.name << " " << animal.food << std::endl;
16   }
17
18 /* For Each Snippet */
19
20   for ( const Animal & animal : zoo ) {
21     std::cout << animal.name << " " << animal.food << std::endl;
22   }
23
```

loops!

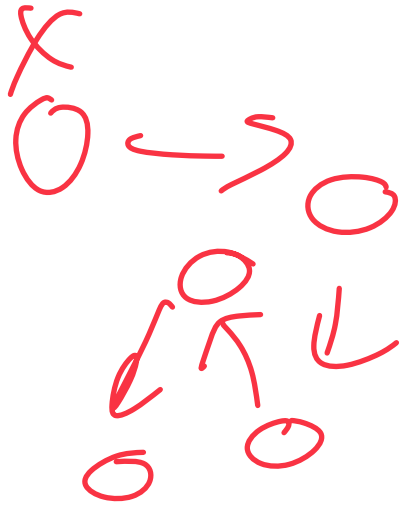
End to beginning

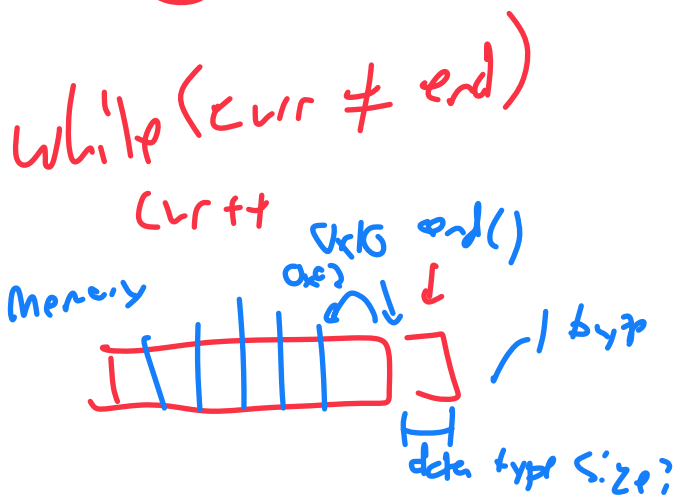
++()

Iterators

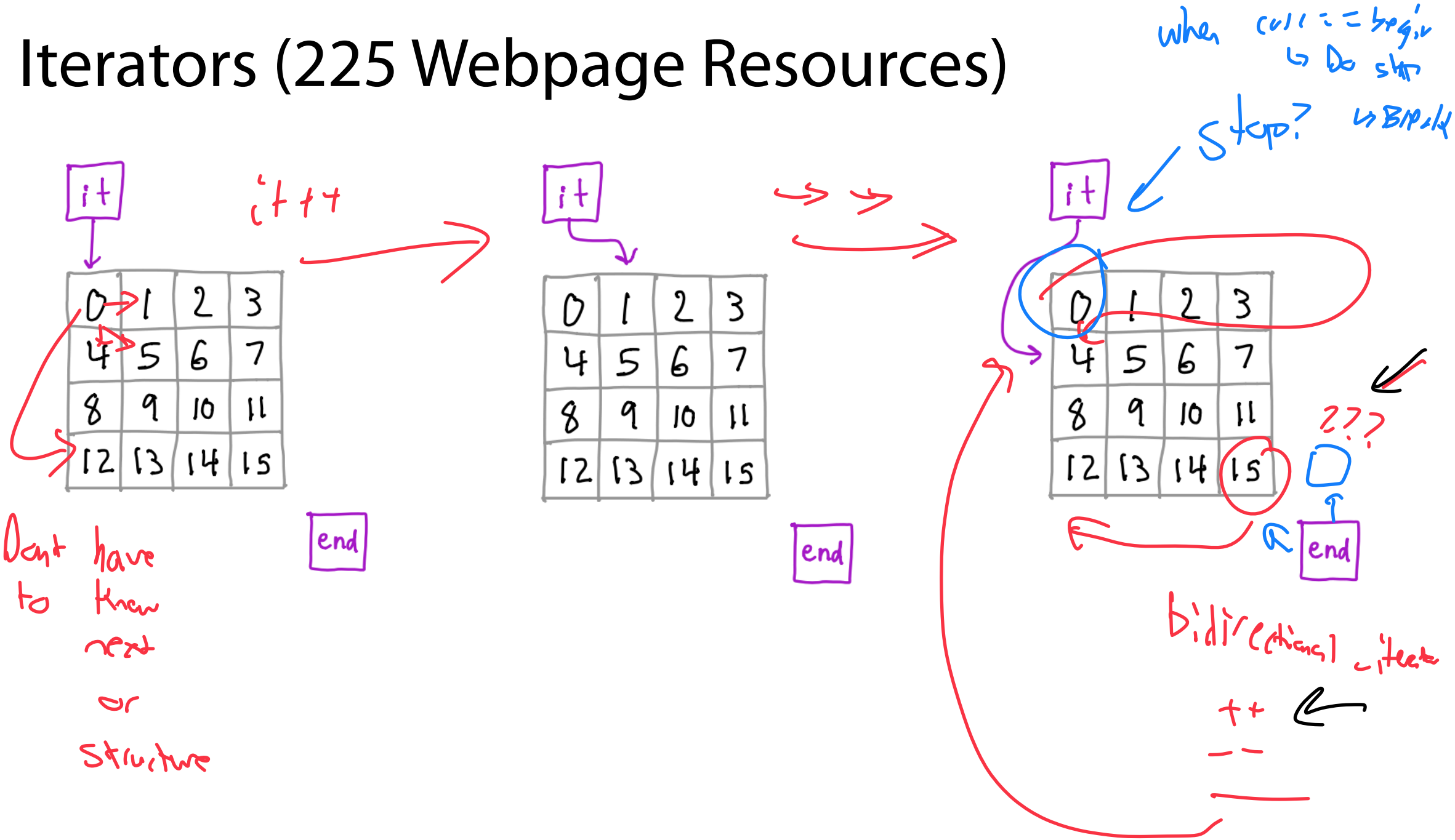
Iterators defined inside a class (and as part of class)

```
 1 template <class T>
 2 class List {
 3
 4     class ListIterator : public
   std::iterator<std::forward_iterator_tag, T> {
 5       public:
 6
 7         ListIterator& operator++();
 8
 9         bool operator!=(const ListIterator& rhs);
10
11         const T& operator*();
12     };
13
14     ListIterator begin() const;
15
16     ListIterator end() const;
17 };
18
19
```



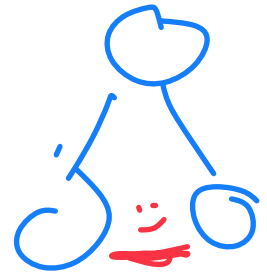


Iterators (225 Webpage Resources)



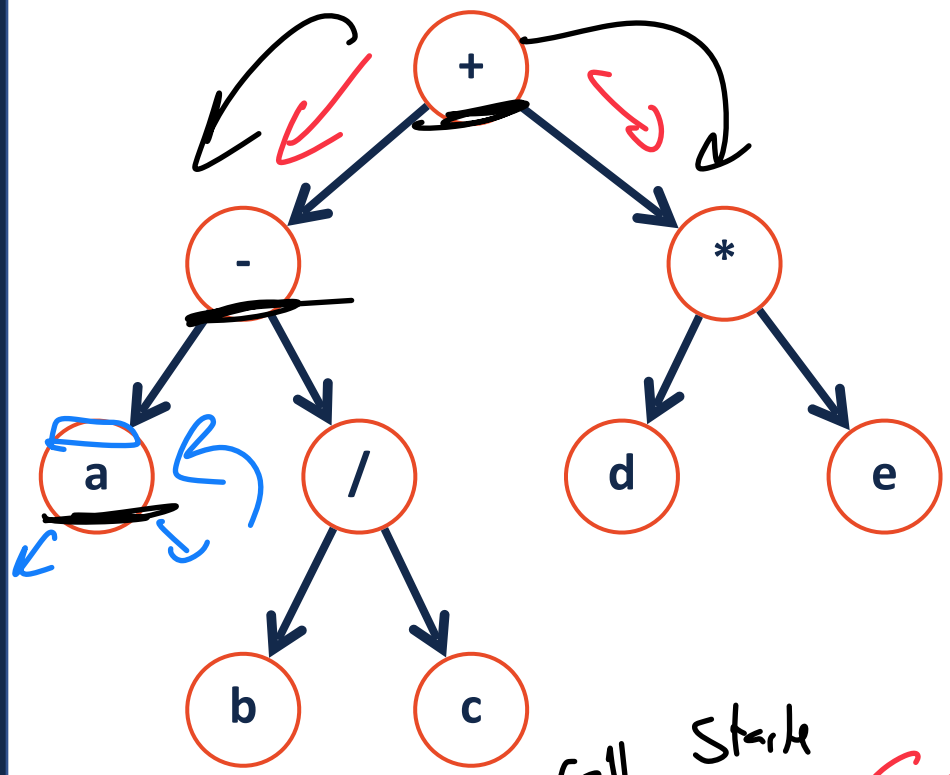
Requested: Tree Traversals

by Iterate through every node in tree → O(n)



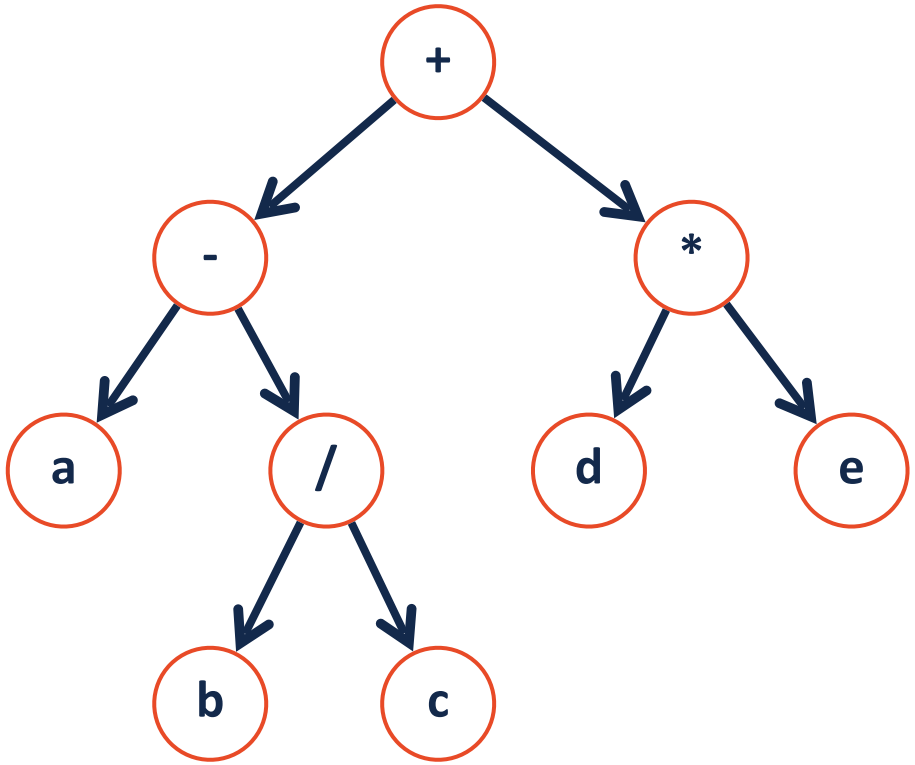
[not needed to traverse ↙ graph trav node + etc

Traversals



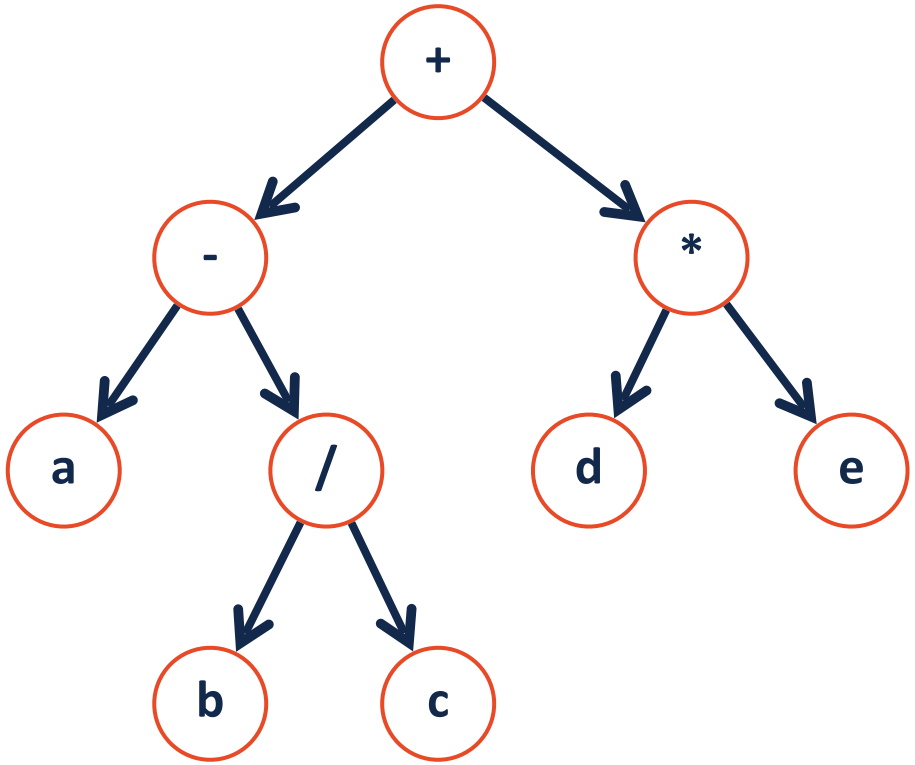

```
1  template<class T>
2  void BinaryTree<T>::postOrder(TreeNode * root)
3  {
4
5    if (root) {
6
7      ____________________________;
8
9      postOrder(root->left);
10
11     ____________________________;
12
13     postOrder(root->right);
14
15     processNode(root)____________;
16
17   }
18
19
20
21 }
```

Traversals




```
1  template<class T>
2  void BinaryTree<T>::_____Order(TreeNode * root)
3  {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21 }
```

Traversals



- Start!

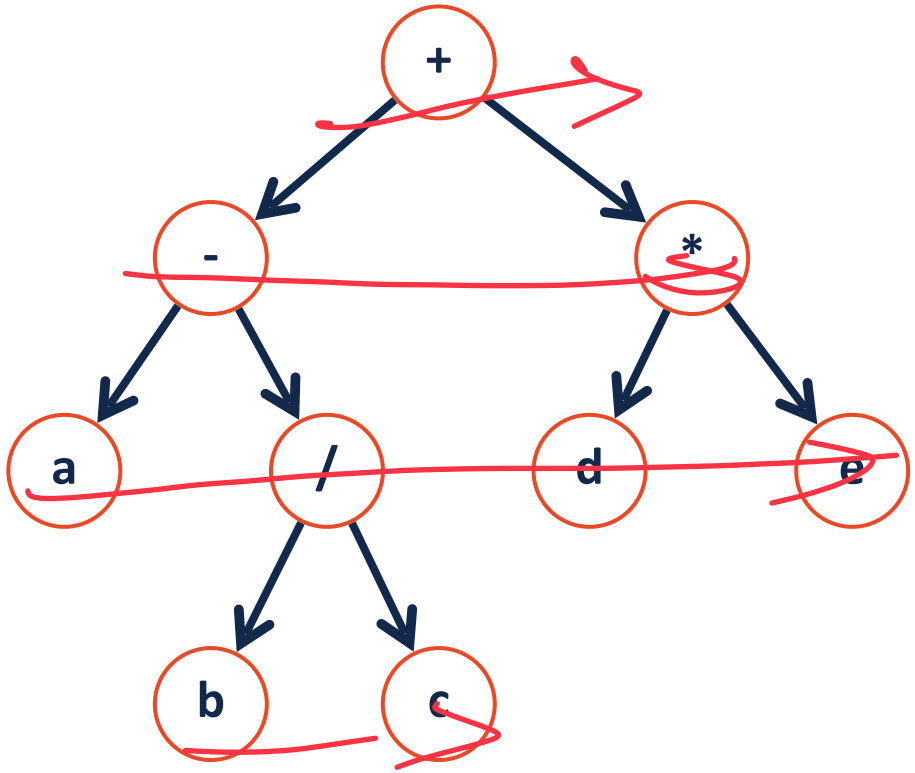
```
1  template<class T>
2  void BinaryTree<T>::_____Order(TreeNode * root)
3  {
4
5    if (root) {
6
7        ____________________________;
8
9        _____Order(root->left);
10
11       ____________________________;
12
13       _____Order(root->right);
14
15       ____________________________;
16
17   }
18
19
20
21 }
```

← Pre

← In orde

← Post

Level-Order Traversal




```
template<class T>
void BinaryTree<T>::lOrder(TreeNode * root)
{

    Queue<TreeNode*> q;
    q.enqueue(root);

    while( q.empty() == False){

        TreeNode* temp = q.head();
        process(temp);

        q.dequeue();

        q.enqueue(temp->left);
        q.enqueue(temp->right);

    }
}
```

Requested: Amortized Analysis

When an algorithm has an infrequent 'costly' step.

Theory

Practice

↳ worst case behavior

Acknowledge

Big O those Data structures are bad

What algorithms *should* we consider from an amortized point of view? (or data structures)

↳ Not as one function call

But as many function call

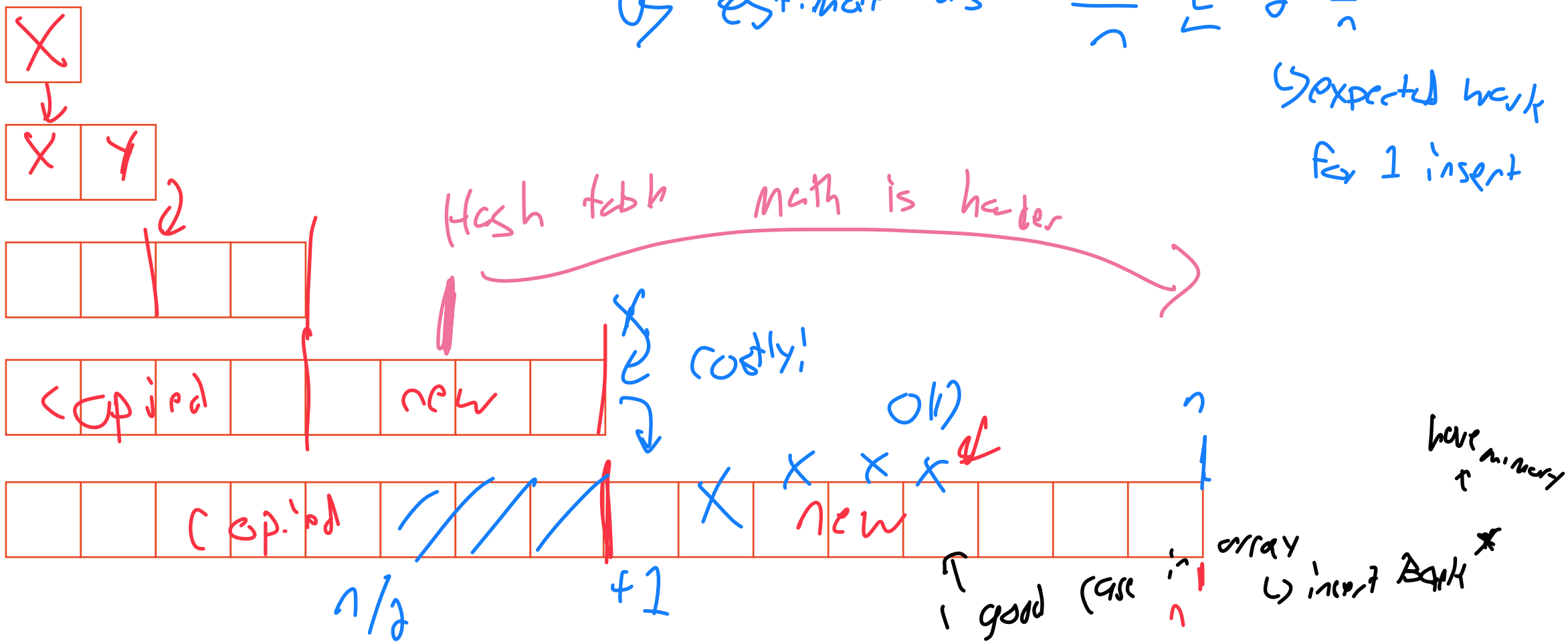
↳ Array amortized

↳ Hash tables / hash map

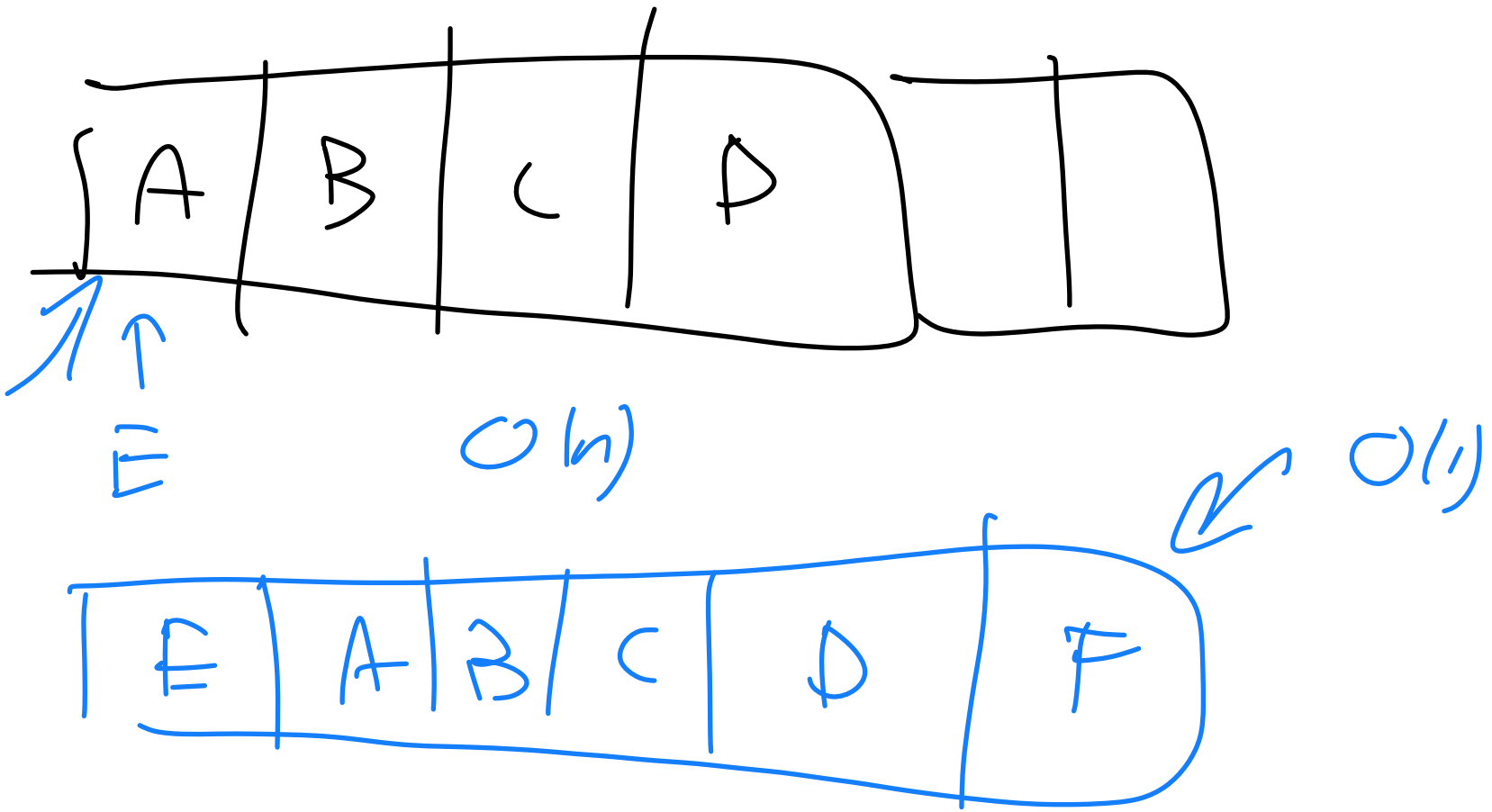
↳ Disjoint set.

Resize Strategy: x2 elements every time

Total copies for n inserts: 2n -1

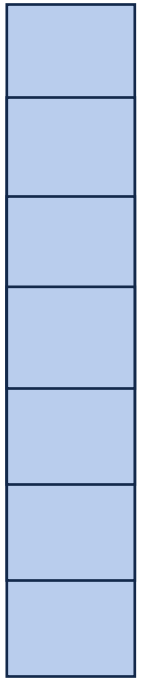


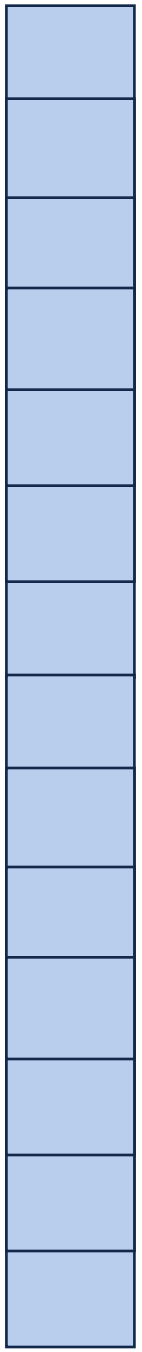
Resize Strategy: x2 elements every time



Resizing a hash table

How do you resize?





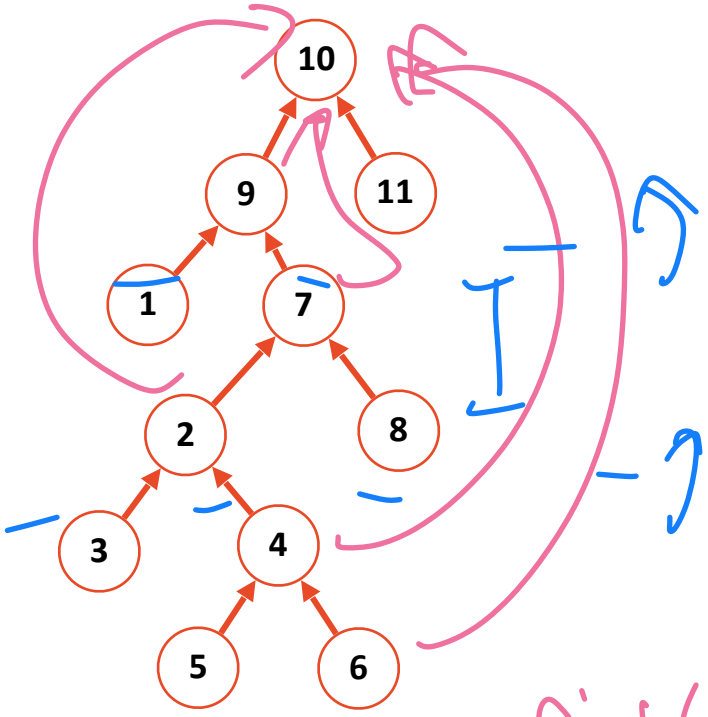
~ 0.7 - 0.9

pseudo-amortized

Amortized Time (Rank w/ Path Compression)

We have **n items** in an Uptree. We make **m find()** calls.

We are interested in the **worst case work** possible **over m calls**.



Expected amortized work is constant

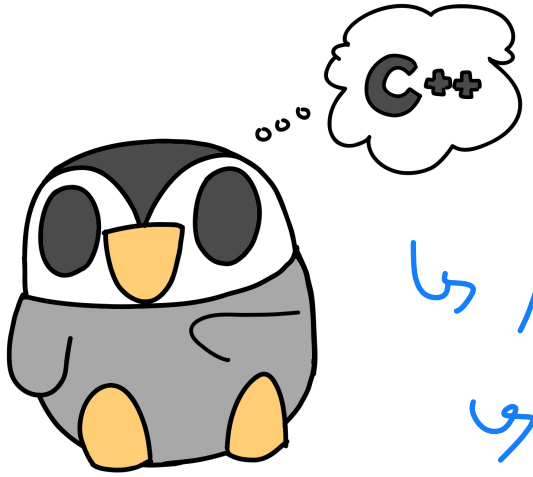
So many calls can produce work

Bad skew

O(1)

find(6) → find(6)

A partial review of topics in CS 225



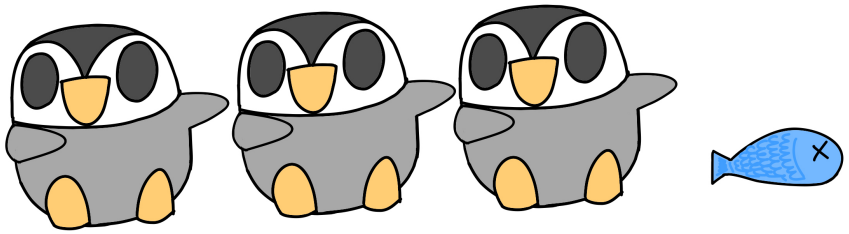
↳ Memory management

↳ keywords

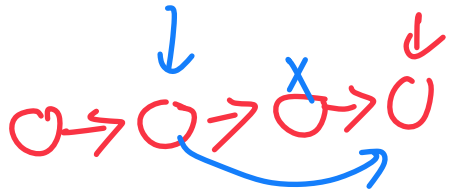
↳ Pointers

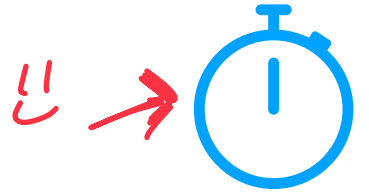
You've used this stuff the entire semester

Lists



Array Implementation



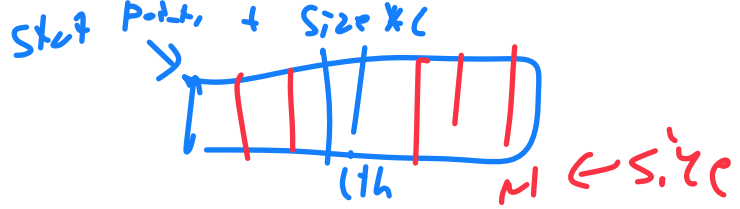


	Singly Linked List	Array
Look up **arbitrary** location ↳ index (random access)	O(n)	O(1)
Insert after **given** element ↳ pointer to object (*p)	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)	O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

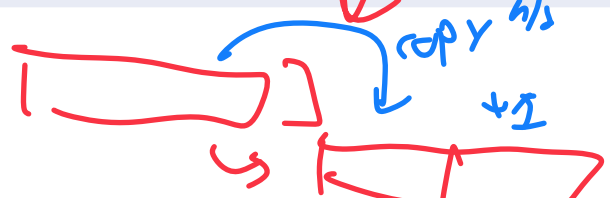
LL. insert (3)

September 6 (Array 2 Lecture)

Array Implementation

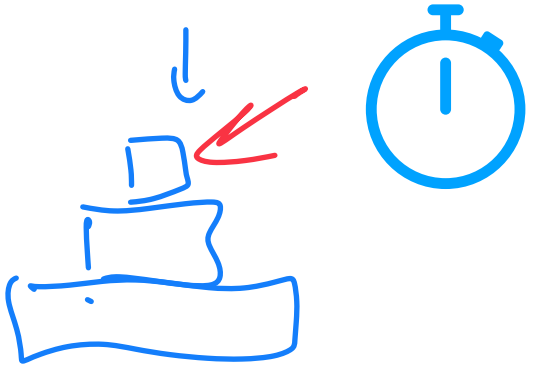


	Singly Linked List	Array
Look up **arbitrary** location Random Access	O(n)	O(1)
Insert after **given** element Given pointer to object	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)	O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)



Stack ADT

- [Order]: last in first out (LIFO)



- [Implementation]: vector / deque ← C++
  LL or array list
- [Runtime]: push() O(1)
  pop() O(1)

trade off → random access

Queue ADT

- [Order]: First in first out FIFO
- [Implementation]: LL w/ head and tail
  Circular array list*
  * worst case for array is O(n) b/c resize
- [Runtime]: enqueue O(1)*
  dequeue

September 11 (Iterator Lecture)

Iterators

The actual iterator is defined as a class **inside** the outer class:

1. It must be of base class **`std::iterator`**

2. It must implement at least the following operations:

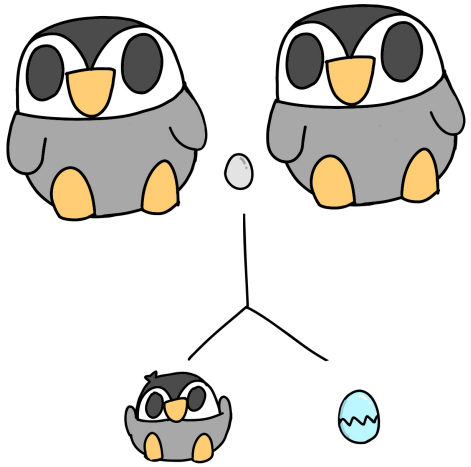
```
Iterator& operator ++()

const T & operator *()

bool operator !=(const Iterator &)
```

```
1
2  std::vector<Animal> zoo;
3
4
5  /* Full text snippet */
6
7    for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
8      std::cout << (*it).name << " " << (*it).food << std::endl;
9    }
10
11
12 /* Auto Snippet */
13
14   for ( auto it = zoo.begin(); it != zoo.end; ++it ) {
15     std::cout << animal.name << " " << animal.food << std::endl;
16   }
17
18 /* For Each Snippet */
19
20   for ( const Animal & animal : zoo ) {
21     std::cout << animal.name << " " << animal.food << std::endl;
22   }
23
24
25
```

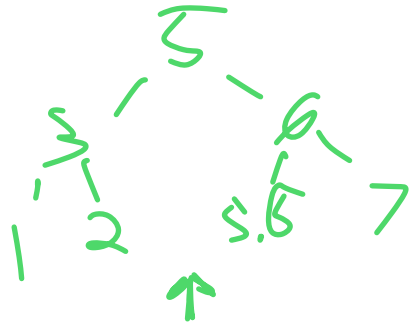
for each animal in zoo
cast Animal

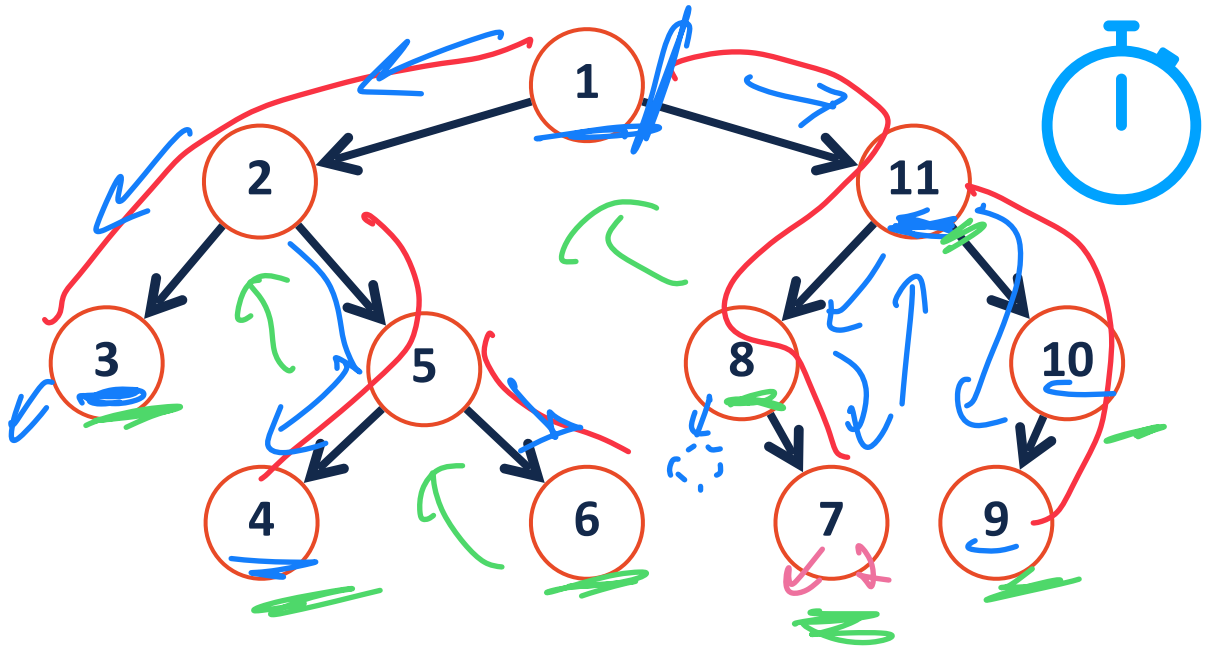
Trees

↳ Not every tree you saw is holy!



Tree Traversals





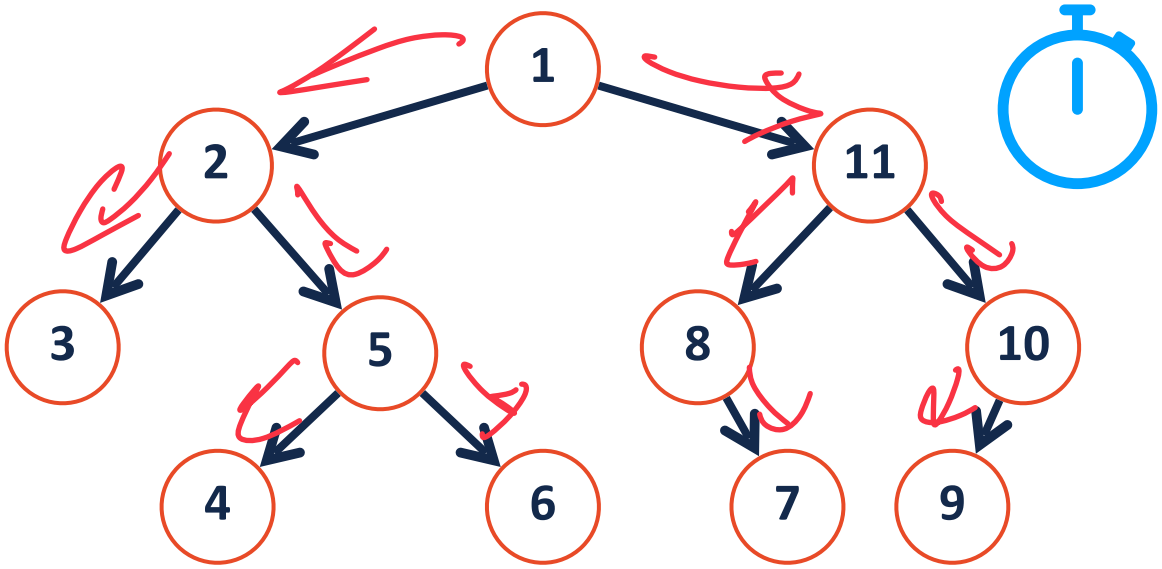
Pre-order: 1 2 3 5 4 6 11 8 7 10 9

In-order: 3 2 4 5 6 1 8 7 11 9 10

left curr right

Post-order: 3 4 6 5 2 7 8 9 10 11 1

Tree Traversals



Pre-order: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

← process node as soon as visit

In-order: 3, 2, 4, 5, 6, 1, 8, 7, 11, 9, 10

← left child first

Post-order: 3, 4, 6, 5, 2, 7, 8, 9, 10, 11, 1

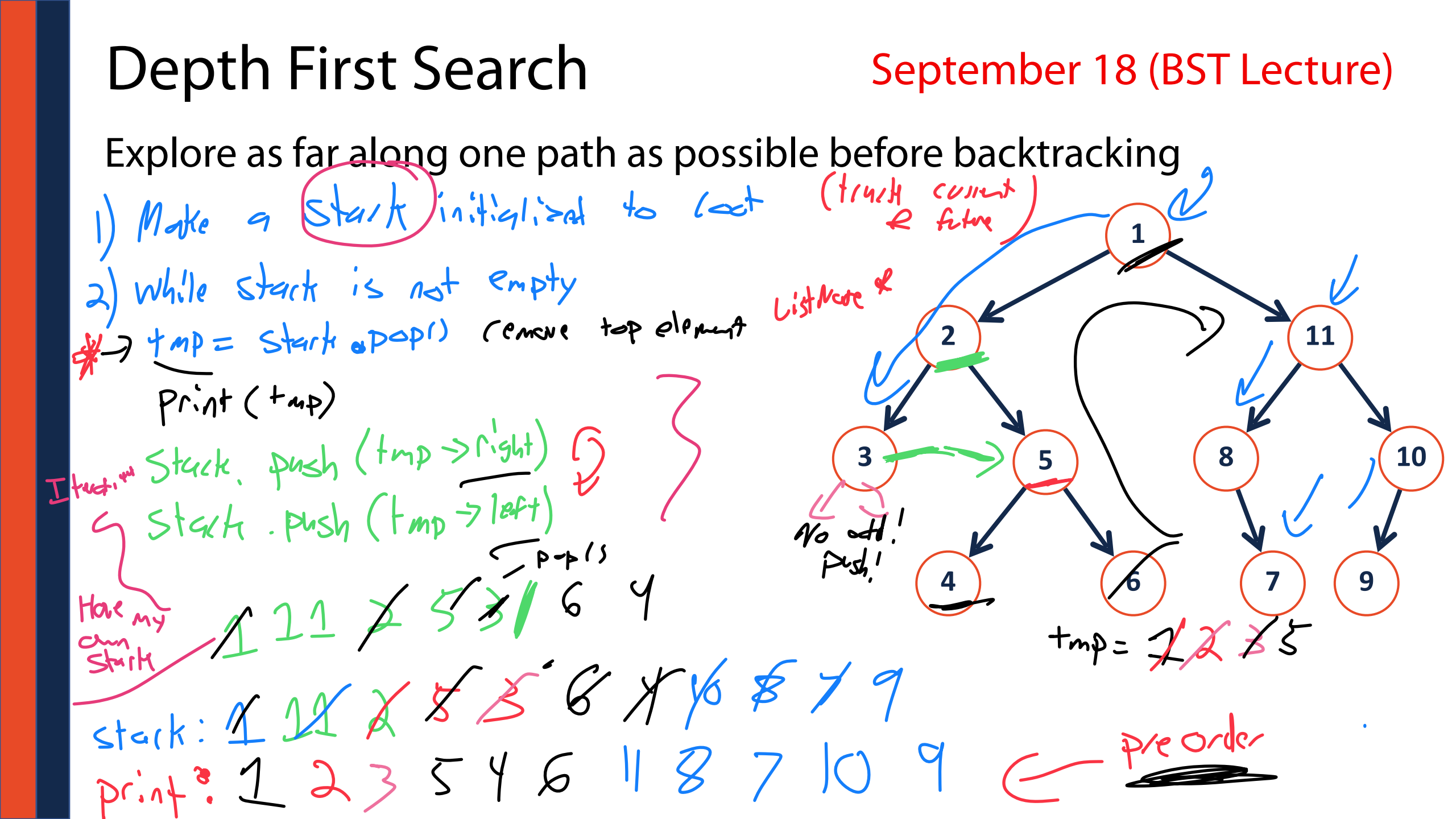
← Both children first

September 15 (Tree Traversal)

Depth First Search

September 18 (BST Lecture)

Explore as far along one path as possible before backtracking



Breadth First Search

Fully explore depth i before exploring depth i+1

pre
recursion
in
recursion
post

1) Make a queue init to root

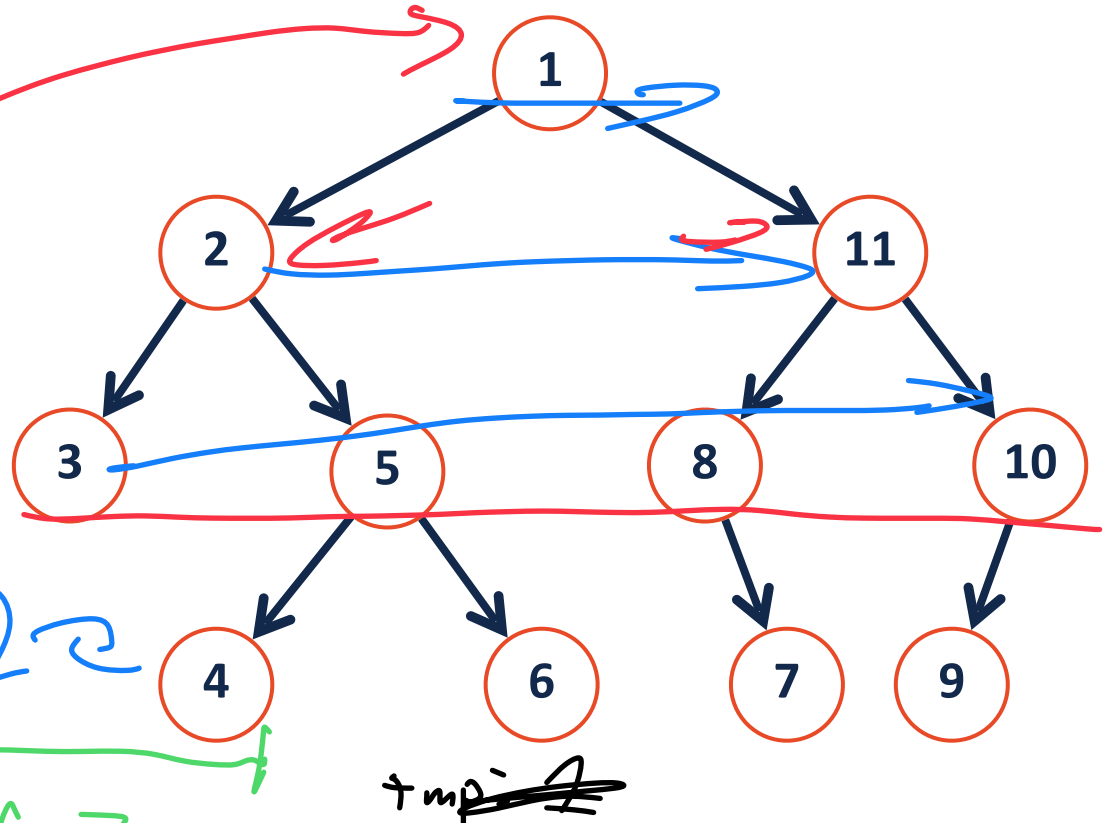
2) while queue is not empty

tmp = queue.dequeue() // free Node *

print tmp

queue.enqueue(tmp->left)

queue.enqueue(tmp->right)



tmp

Locked entire level

front

Queue: ~~1~~ ~~2~~ ~~11~~ ~~3~~ ~~5~~ ~~8~~ ~~10~~ 4 6 7 9

print: 1 2 11 3 5 8 10 4 6 7 9

BST Find

To insert or remove I need find

1) Start at root node

1.5) check if root is null → if so return root;

2) Check value of root->key

↳ if root->key == query

↳ if yes, return root

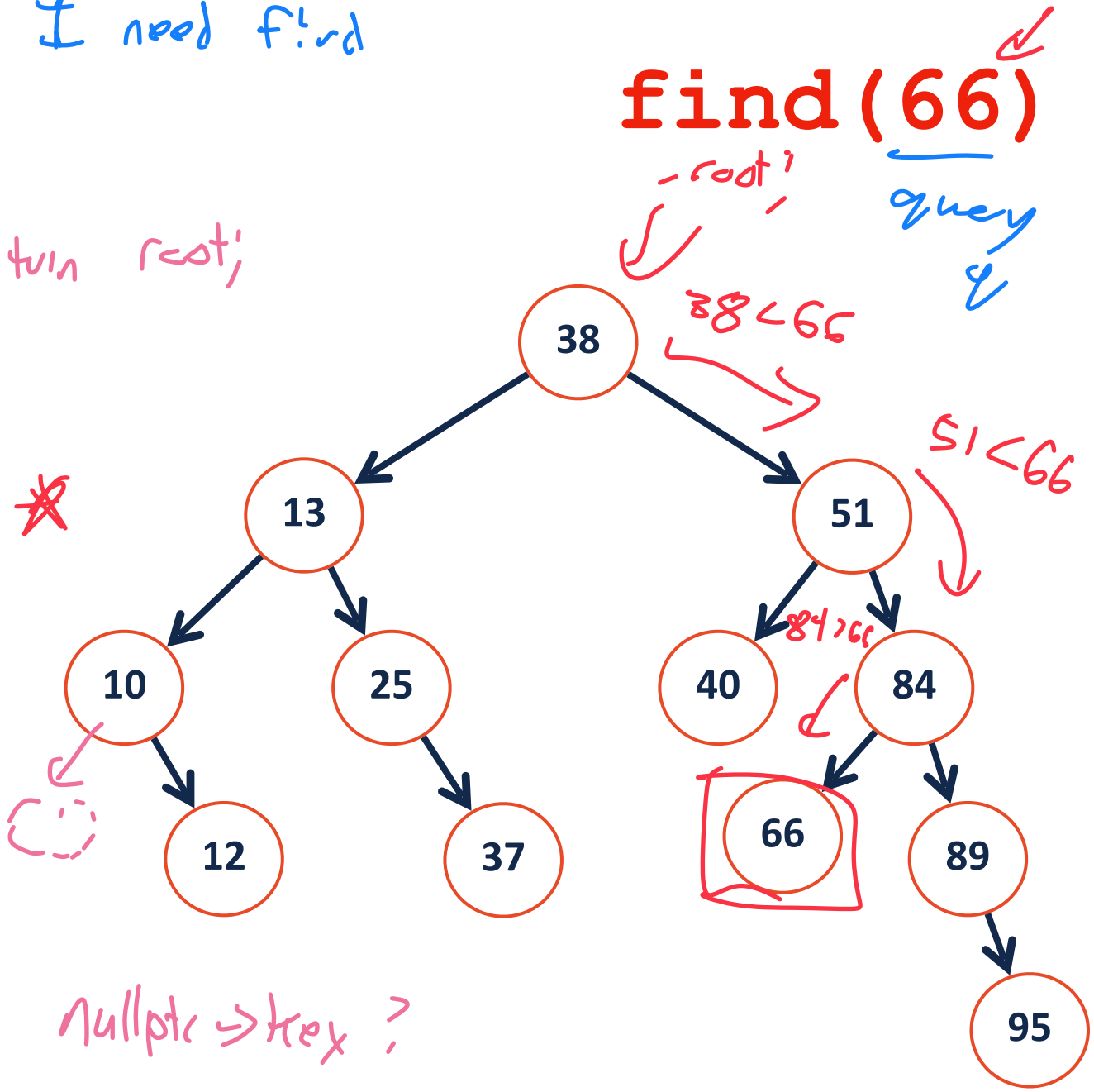
if root->key < query

recurse right

if root->key > query

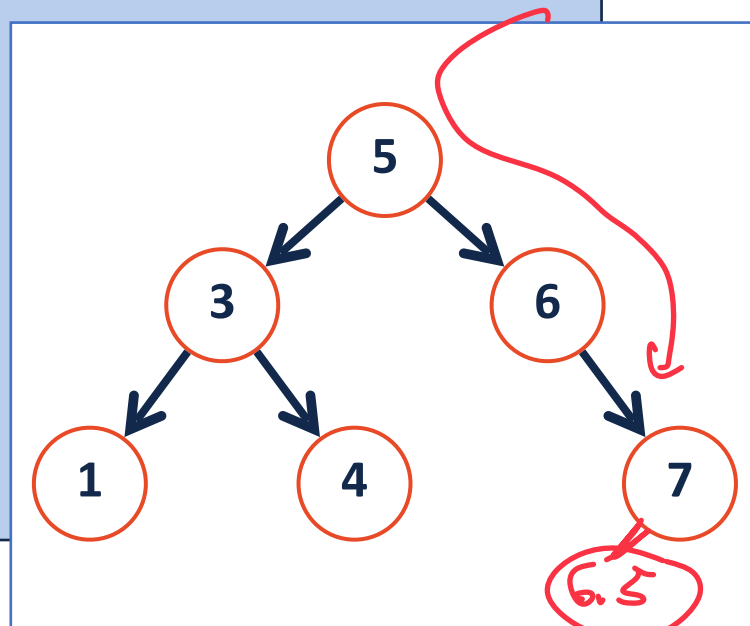
recurse left

*



```
1  template<typename K, typename V>
2
3  void _insert(const K & key, const V & val) {
4
5      return _insert(root, key, val);
6  }
7
```

```
1  template<typename K, typename V>
2
3  void _insert(TreeNode *& root, const K & key, const V & val) {
4
5      1) Do a find ( )
6
7      2) Insert new Node
8
9
10
11
12
13
14
15
16 }
```



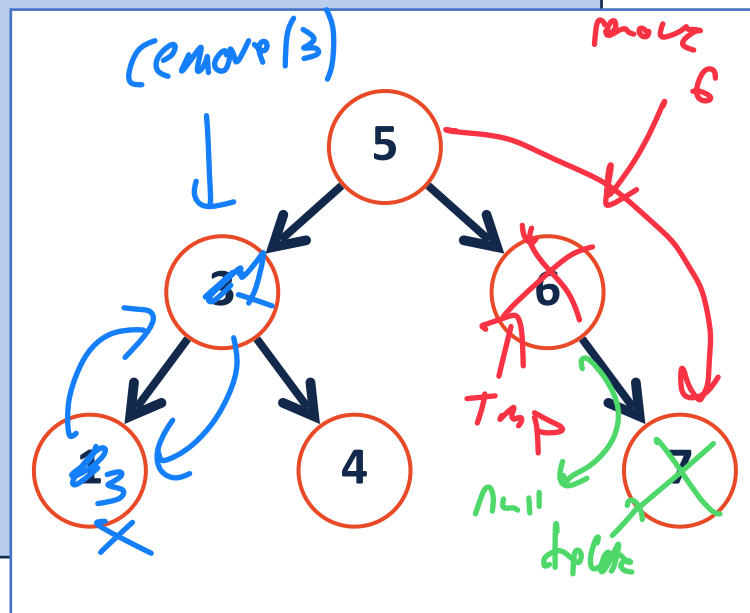
Review material at start of class

```
1  template<typename K, typename V>
2
3  void _remove(TreeNode *& root, const K & key) {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 }
```

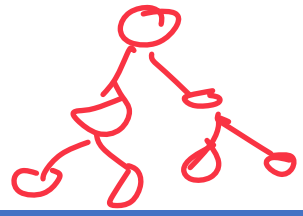
1) find (6) ← *&

2) 3 cases:

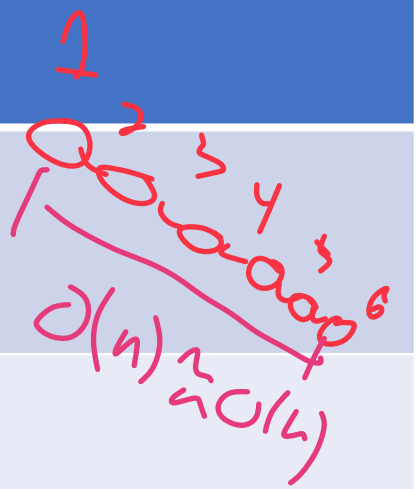
- 0 - child remove
- 1 - child removal
  - ↳ LL remove
- 2 - child removal
  - ↳ find IOP / IOS
  - ↳ swap IOP w/ root
  - ↳ remove (key)



BST Analysis – Running Time



Operation	BST Worst Case
find	O(h) → where h is height h is O(n)
insert	find O(h) insert O(1) → O(h)
remove	find (O(h)) → remove obj + find (O(h)) → IOP/IOS + 1 O(h) ∴
traverse	O(n) ← O(n)



BST Analysis – Running Time

Operation	BST Worst Case
find	O(h) = O(n)
insert	O(h) = O(n)
remove	O(h) = O(n)
traverse	O(n)

AVL Rotations

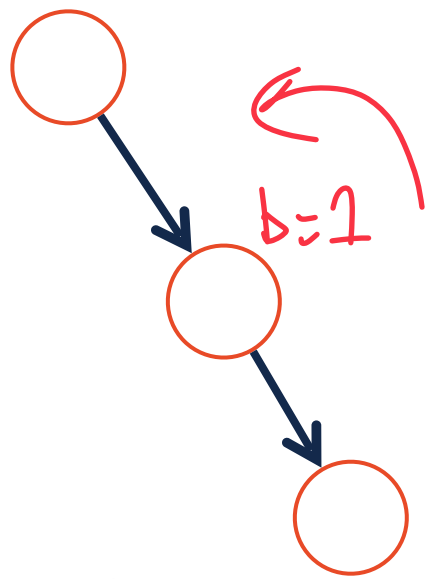
O(1) operations
Reduce height by 1

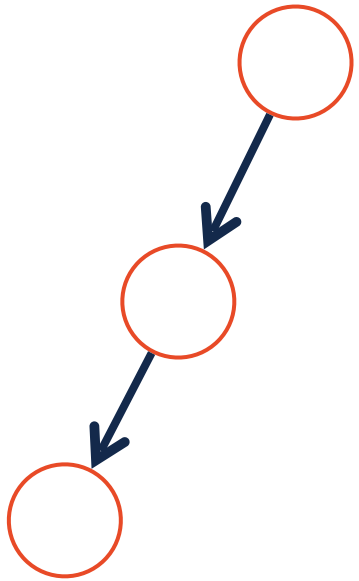
Very important!

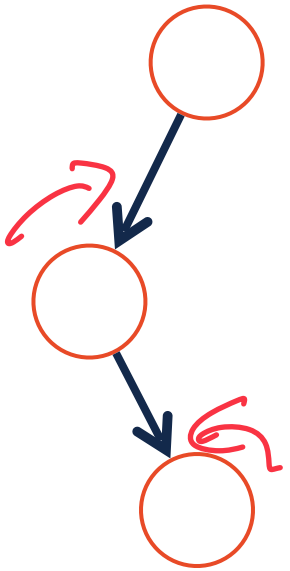
b = 2

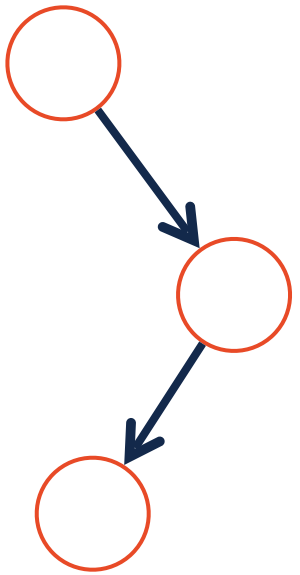
Simple

Complex









2, 2

-2, -1

Left

Right

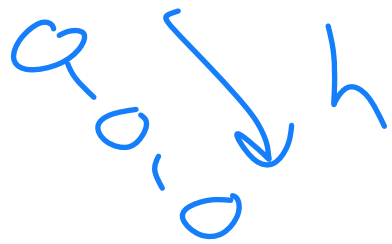
-2, 1

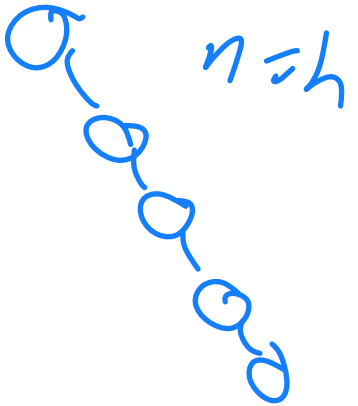
2, -2

Left Right

Right Left

AVL Tree Analysis





For an AVL tree of height h:

Find runs in: O(h).

Insert runs in: O(h).

Remove runs in: O(h).

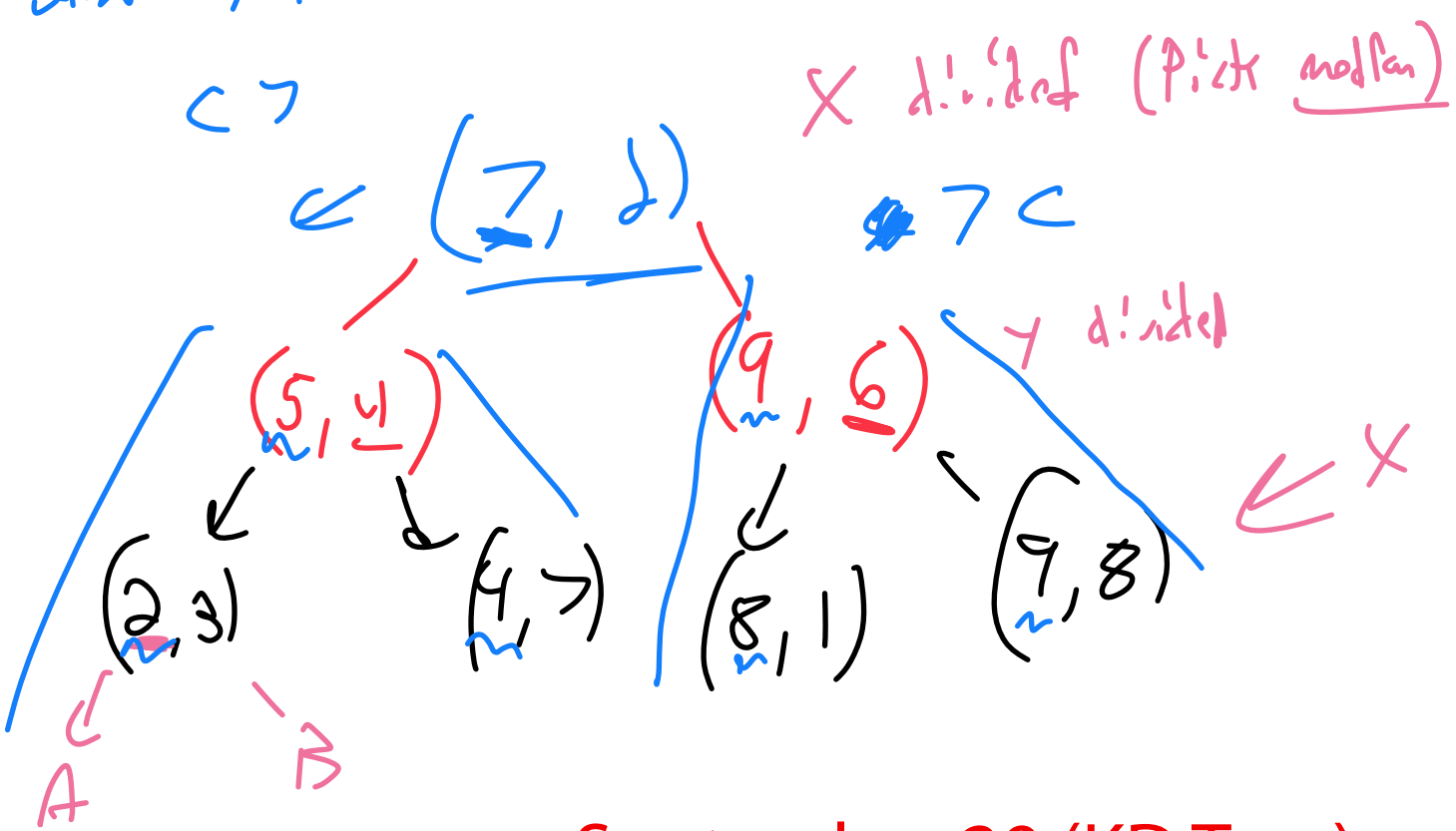
Claim: The height of the AVL tree with n nodes is: O(log n).

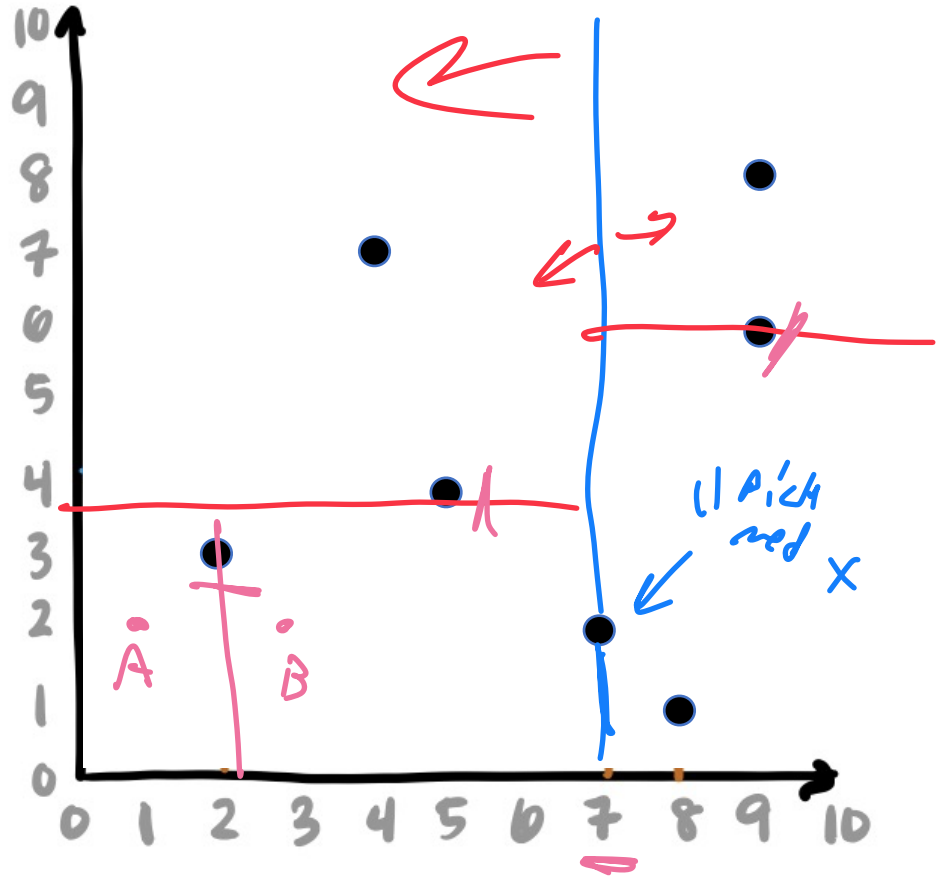
September 27 (AVL analysis)

Nearest Neighbor: k-d tree

A **k-d tree** is similar but splits on points:

(7,2), (5,4), (9,6), (4,7), (2,3), (8,1), (9,8)





BTree Properties

A **BTrees** of order **m** is an m-ary tree and by definition:

- All keys within a node are ordered
- All nodes contain no more than **m-1** keys.
- All internal nodes have exactly **one more child than keys**

Root nodes can be a leaf or have $[2, m]$ children.

All non-root, internal nodes have $[\lceil \frac{m}{2} \rceil, m]$ children.

All leaves in the tree are at the same level.

Heaps

(min)Heap

October 13 (Heaps)

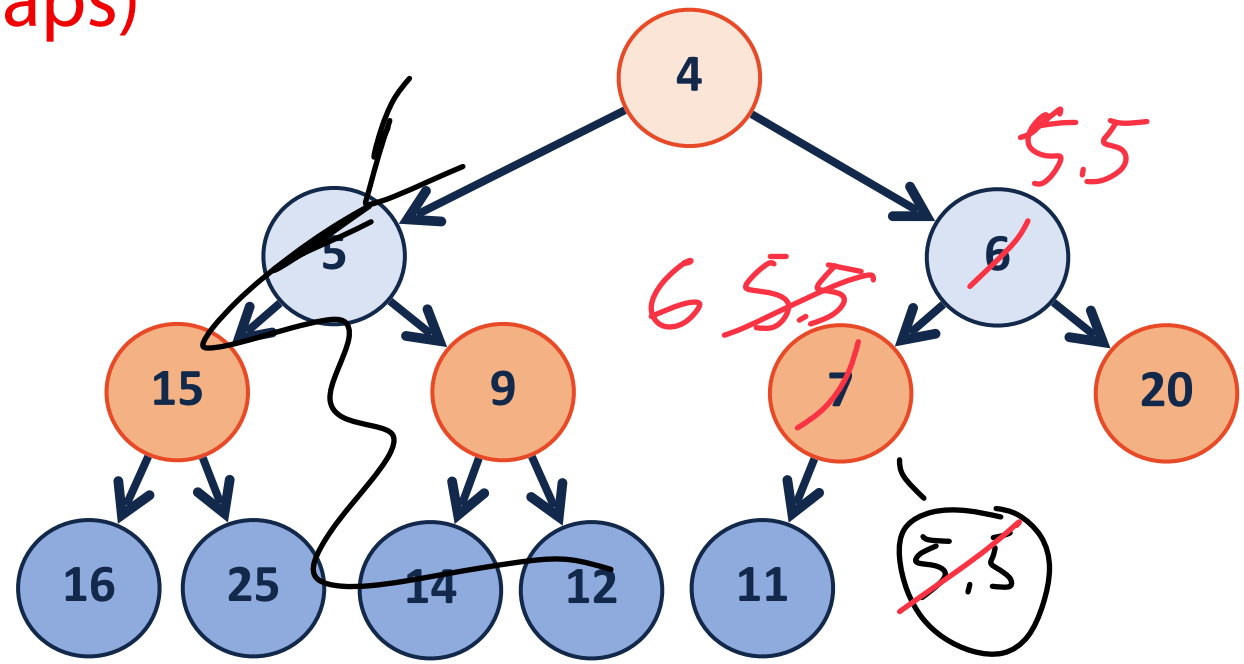
Insert (i)

↳ Array pushBack (i)

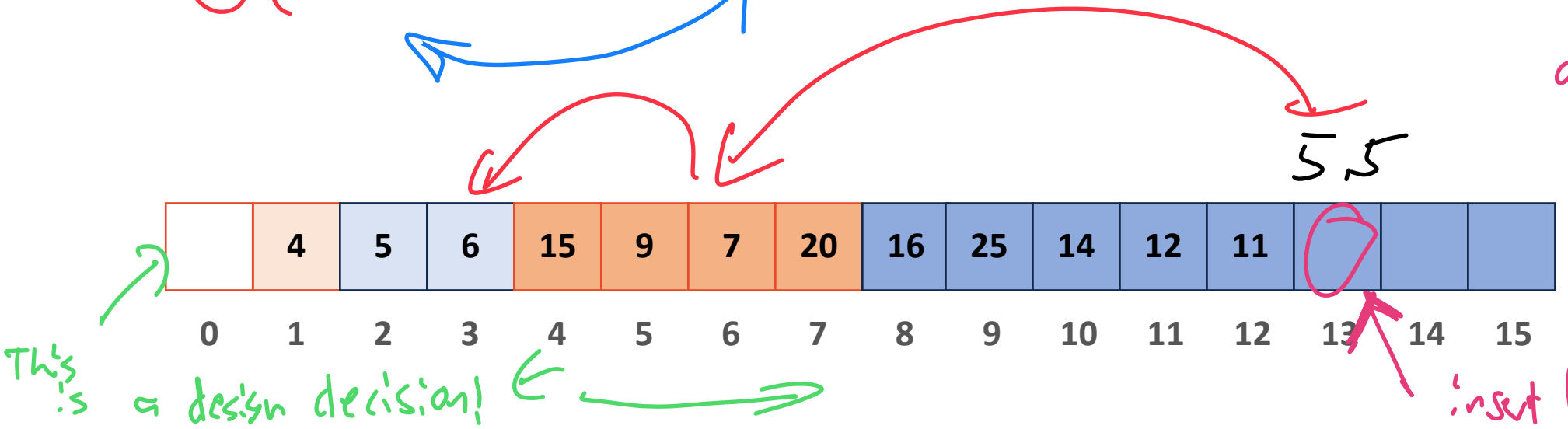
↳ swap until heap again

↳ 'heapify up'

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



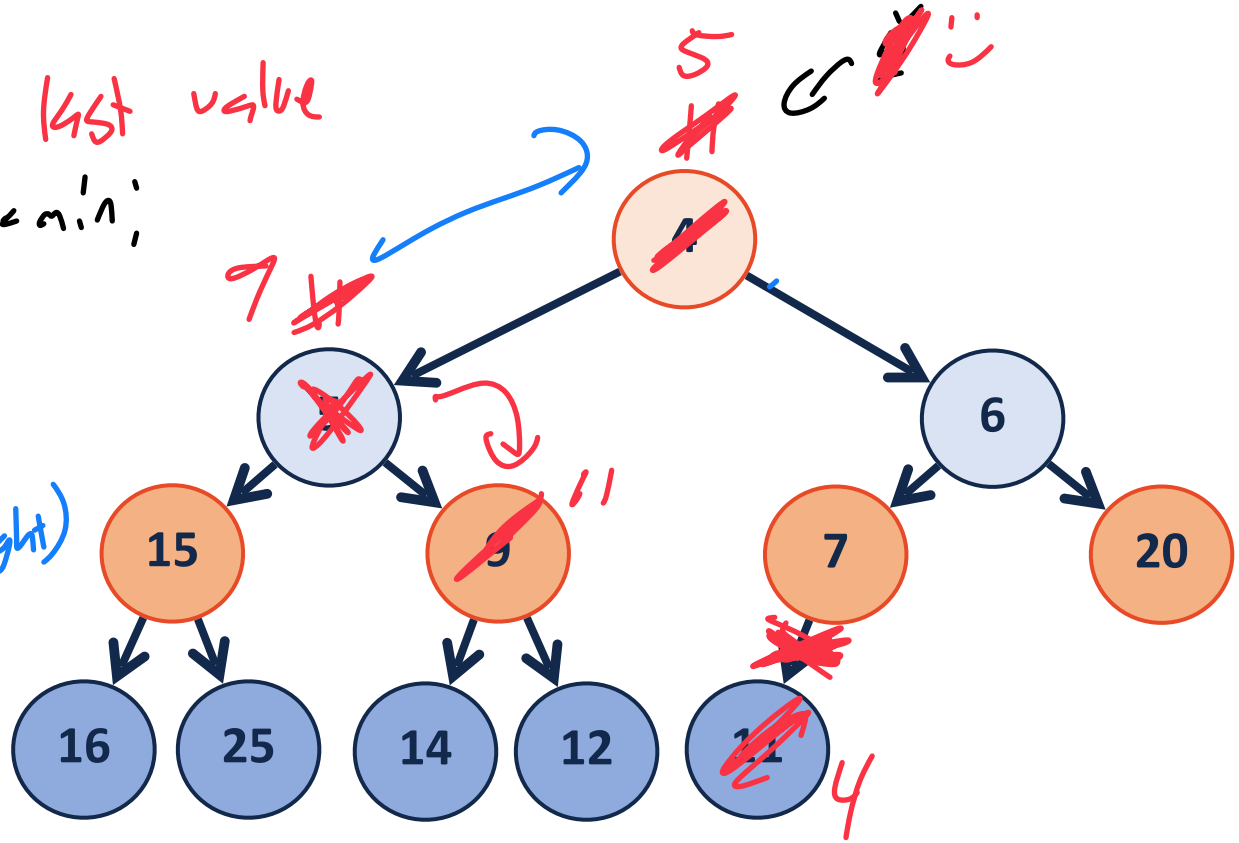
complete tree is balanced

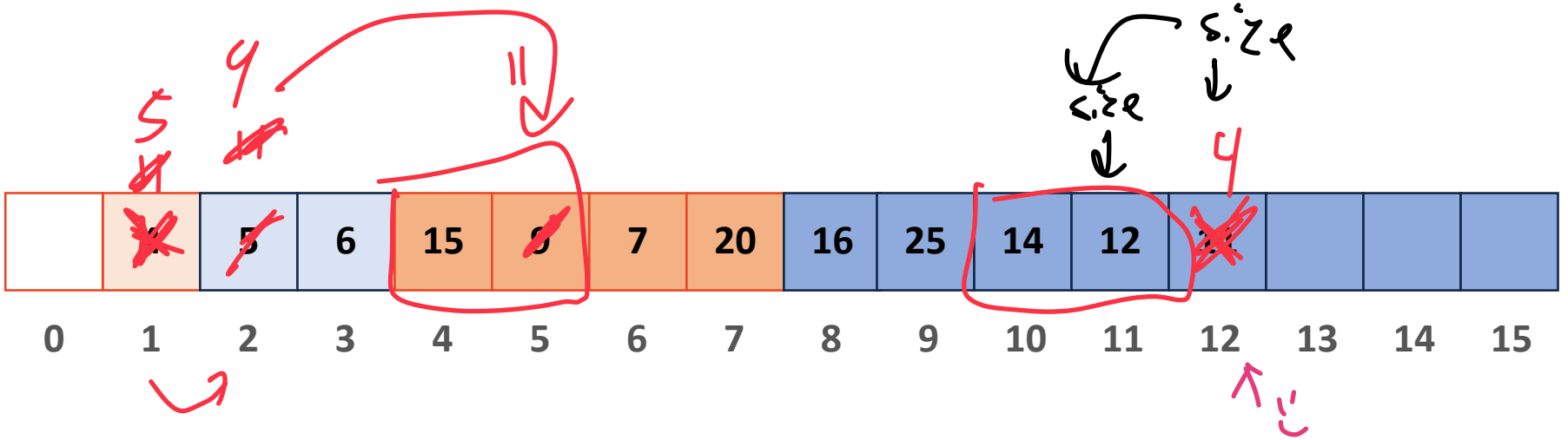


array stores a tree

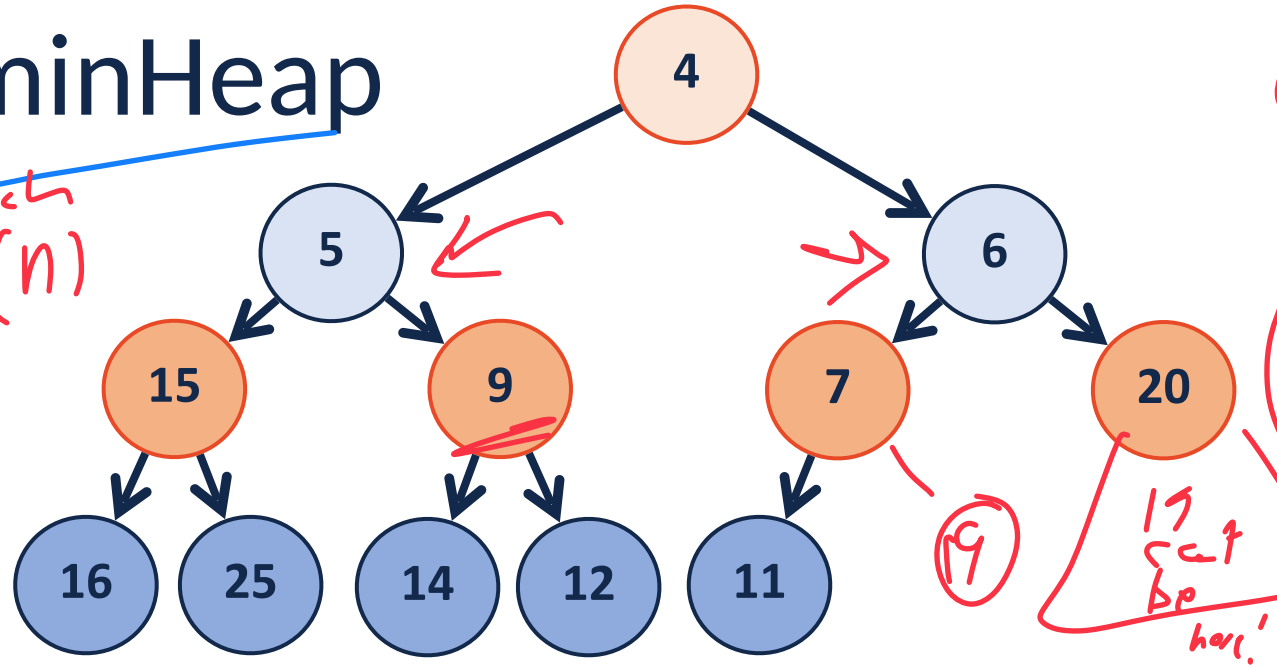
removeMin

1) Swap min value (root) w/ last value → and return tempmin;
2) Delete min by size--
3) heapifyDown (root)
   ↳ Swap root w/ min (left, right) recursively





minHeap

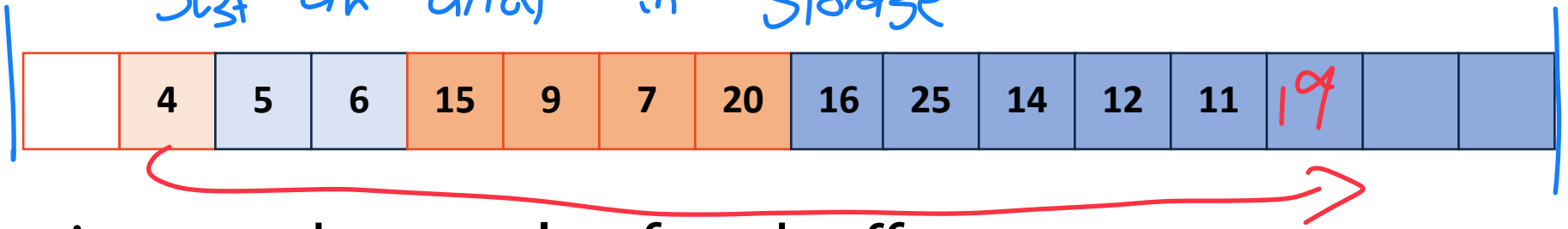


1. Construction → O(n)

2. Insert → O(log n)

3. RemoveMin → O(log n)

Just an array in storage



minHeap is a good example of tradeoffs:

Nearly optimal * for every function ⟷ Cost of access!

Not suitable for random access of find

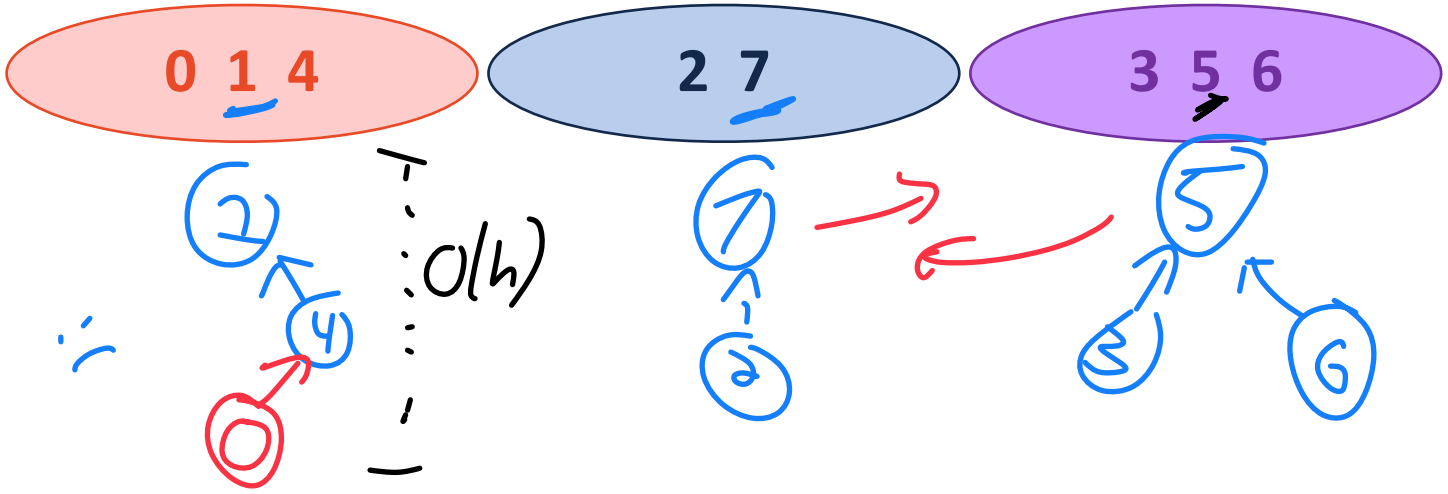
October 16 (EC Project / Disjoint Sets)

Disjoint Sets

Implementation #2

October 18 (Sets 2)

union(5, 7)



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

Find(k): O(h)

↳ last class: O(1) w/ array

↳ A little slower

Union(k_1, k_2): O(1) b/c we draw one arrow (we choose one value)

↳ last class: O(n)

Uptree

1) All non-canonical elements point to canonical (or 'root')

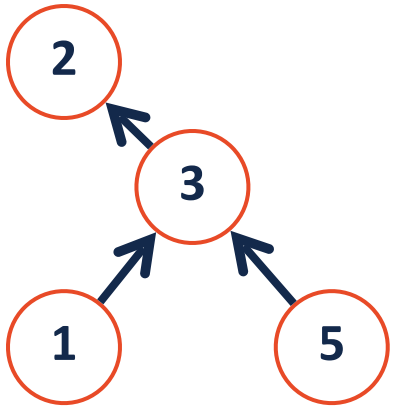
2) Canonical (root) elements store -1

Disjoint Sets Representation

We can represent a disjoint set as an array where the key is the index

The values inside the array stores our sets as a pseudo-tree (UpTree)

- The value **-1** is our representative element (the root)
- All other set members store the index to a parent of the UpTree



Disjoint Sets Find

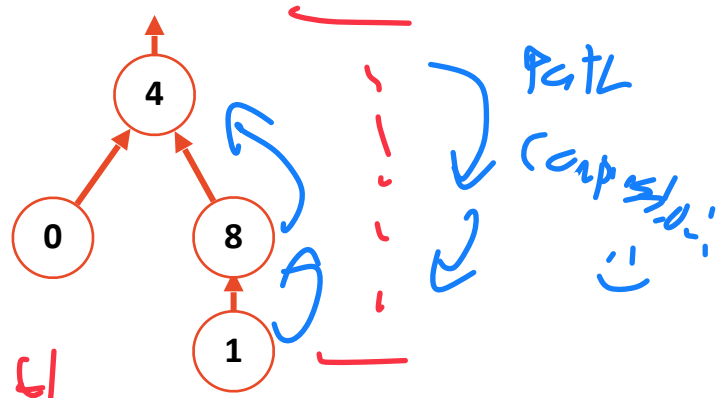
Find(1)

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

-1 is canonical

recurse up

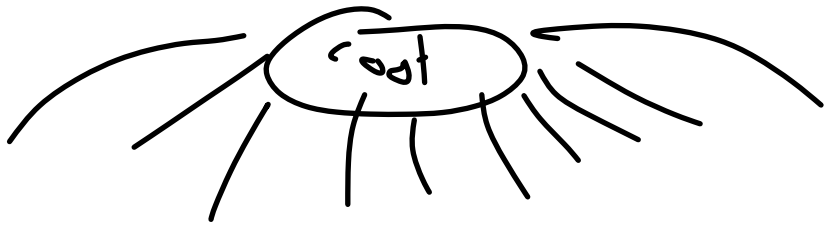
0 1 4 8



Running time? O(h)

What is ideal UpTree?

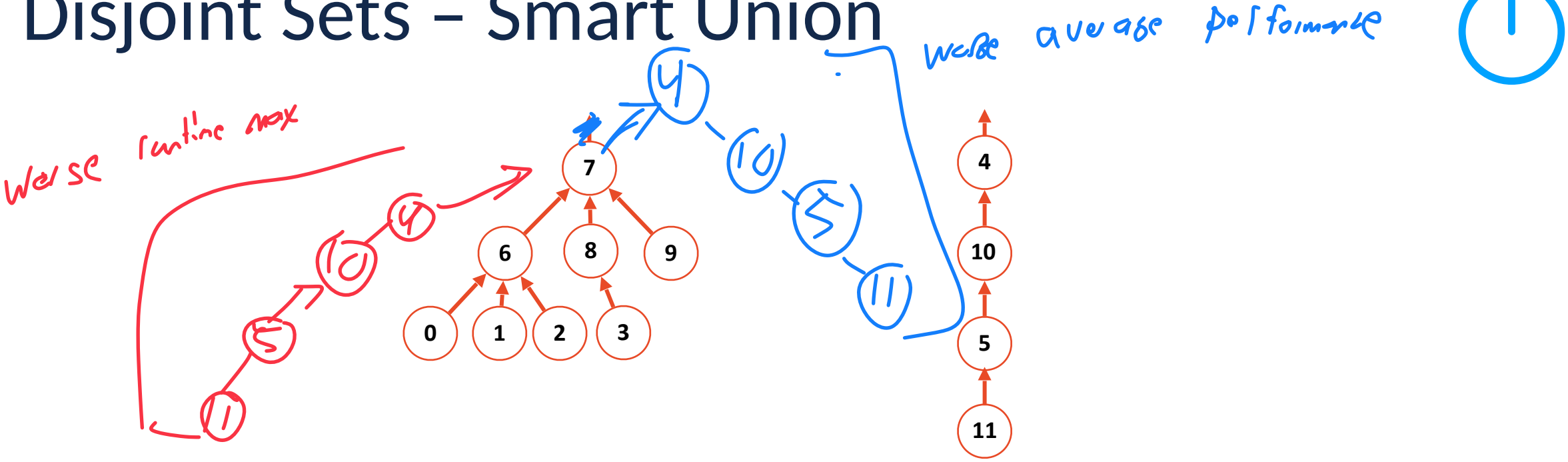
height 1 or 0



return 4

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

Disjoint Sets – Smart Union



Union by height

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

***Idea**: Keep the height of the tree as small as possible.*

Union by size

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

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

Claim that both guarantee the height of the tree is: $\log_2 n$. !!

Final Result

October 23 (Probability in CS)

We have **n items** in an Uptree. We make **m find()** calls. Total work is:

cost between buckets

cost inside buckets

$m \cdot \log^* n \quad + \quad n \cdot \log^* n$

$(m+n) \log^* n$

Eventually my finds are just this

root

Probability in CS

Probabilistic Data Structures

A Hash Table based Dictionary

User Code (is a map):

```
1  Dictionary<KeyType, ValueType> d;
2  d[k] = v;
```

A **Hash Table** consists of three things:

1. A hash function

Data → int

2. A data storage structure

Array

3. A method of addressing *hash collisions*

Hash Table (Separate Chaining)

October 27 (Hashing)

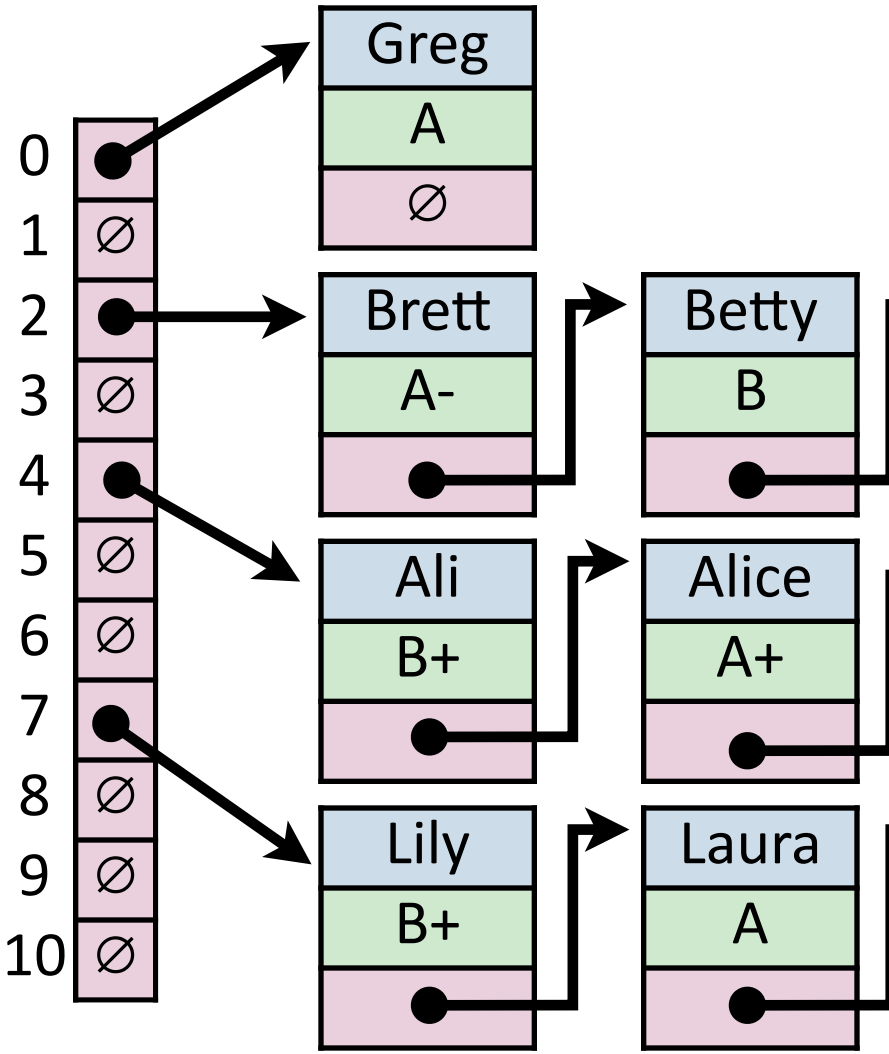
For hash table of size *m* and *n* elements:

Find runs in: O(n) c

Insert runs in: O(1) :)

Remove runs in: O(n)

:(B/c of collisions! ← :(



Simple Uniform Hashing Assumption

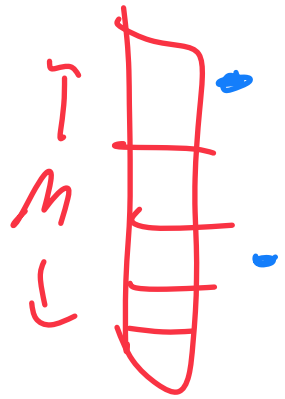
October 30

Given table of size m, a simple uniform hash, h, implies

$$\forall k_1, k_2 \in U \text{ where } k_1 \neq k_2,\ Pr(h[k_1] = h[k_2]) = \frac{1}{m}$$

Uniform: All keys equally likely to hash any value

$Pr[h(k_1) = X] = \frac{1}{m}$



Independent: All keys hash independently of each other

Separate Chaining Under SUHA

Under SUHA, a hash table of size *m* and *n* elements:

Find runs in: $1 + \alpha$ $1 + \alpha + 1/m$.

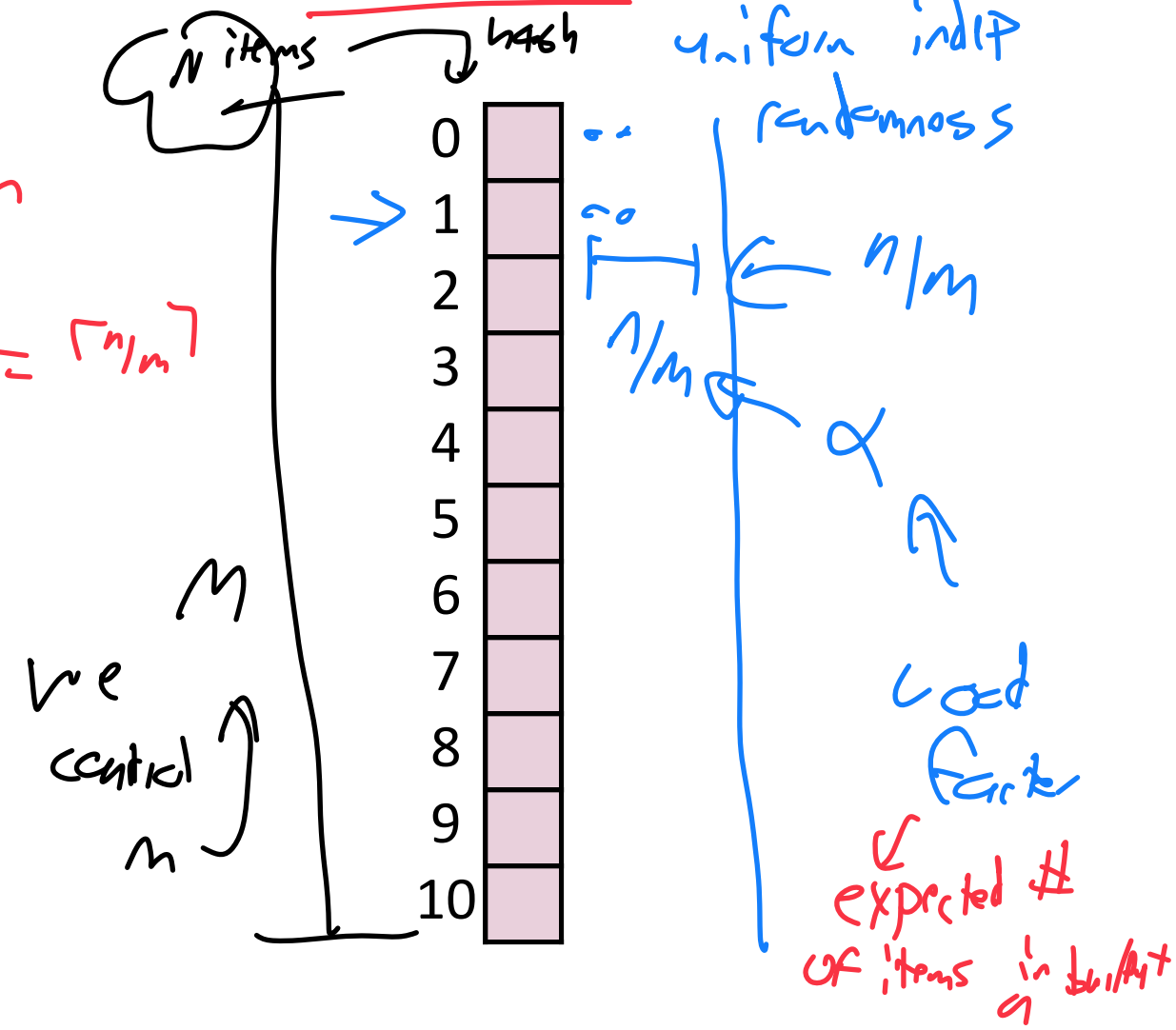
constants

Insert runs in: $O(1)$.

$\alpha = \lceil n/m \rceil$

Remove runs in: $1 + \alpha$.

Expectations Not Big O



Running Times *(Don't memorize these equations, no need.)*

The expected number of probes for find(key) under SUHA

Linear Probing:

- Successful: **½(1 + 1/(1-α))**
- Unsuccessful: **½(1 + 1/(1-α))²**

Double Hashing:

- Successful: **1/α * ln(1/(1-α))**
- Unsuccessful: **1/(1-α)** ← 1/1-1 = 1/0 !!

Separate Chaining:

- Successful: **1 + α/2**
- Unsuccessful: **1 + α**

OH: $\alpha \to \infty$

CH: $0 \leq \alpha < 1$ (Don't let it be 1)

Instead, observe:

- As α increases:

Search time → ∞

- If α is constant:

↳ Runtime is constant

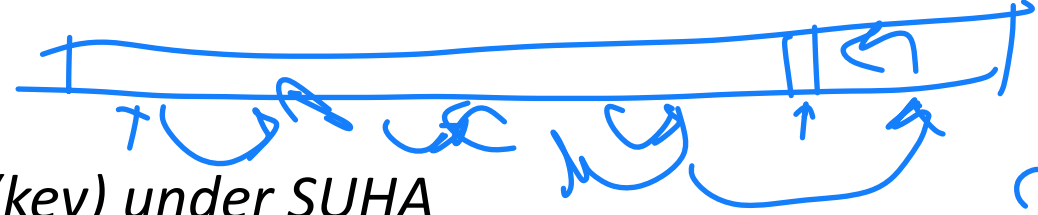
IF n=100, m > 100

m = 200 :)

November 1 (Bloom Filter)

Running Times

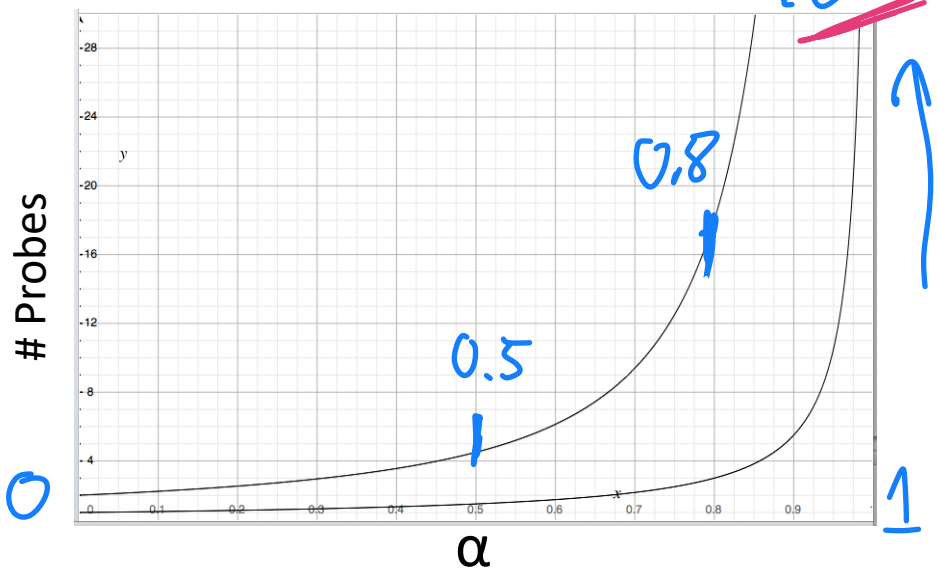
The expected number of probes for find(key) under SUHA

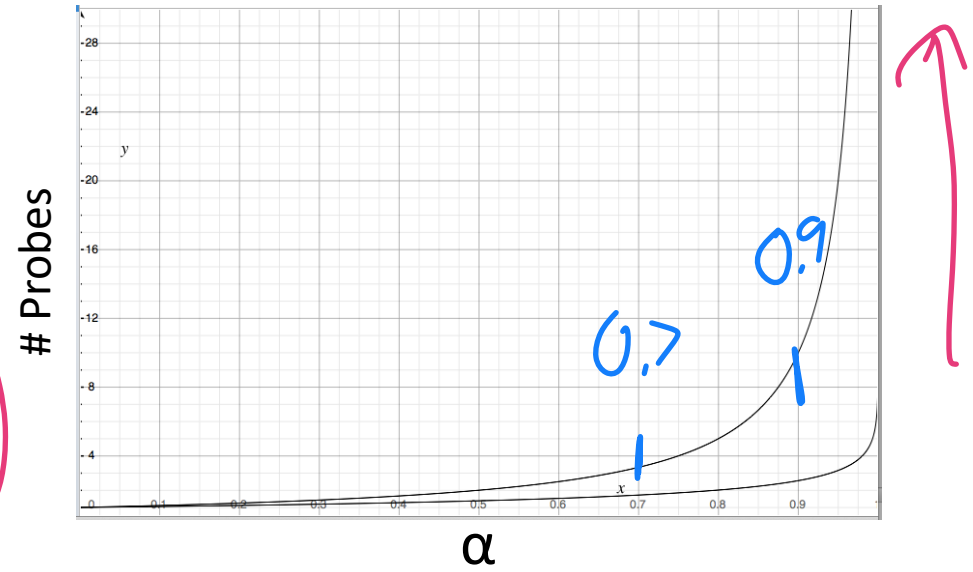


- **Linear Probing:**
  - Successful: **½(1 + 1/(1-α))**
  - Unsuccessful: **½(1 + 1/(1-α))²**

- **Double Hashing:**
  - Successful: **1/α * ln(1/(1-α))**
  - Unsuccessful: **1/(1-α)**

When do we resize?





Running Times

	Hash Table	AVL	Linked List
Find	Expectation*: $O(1)$ Worst Case: $O(n)$	$O(\log n)$	$O(n)$
Insert	Expectation*: $O(1)$ Worst Case: $O(n)$	$O(\log n)$	$O(1)$
Storage Space	$O(m) \sim O(n)$	$O(n)$	$O(n)$

control this

Bloom Filter

November 3 (BF 2)

A probabilistic data structure storing a set of values

$H = \{h_1, h_2, \ldots, h_k\}$

Built from a bit vector of length m and k hash functions

0
0
1
0
0
1
0
1
0
0

m

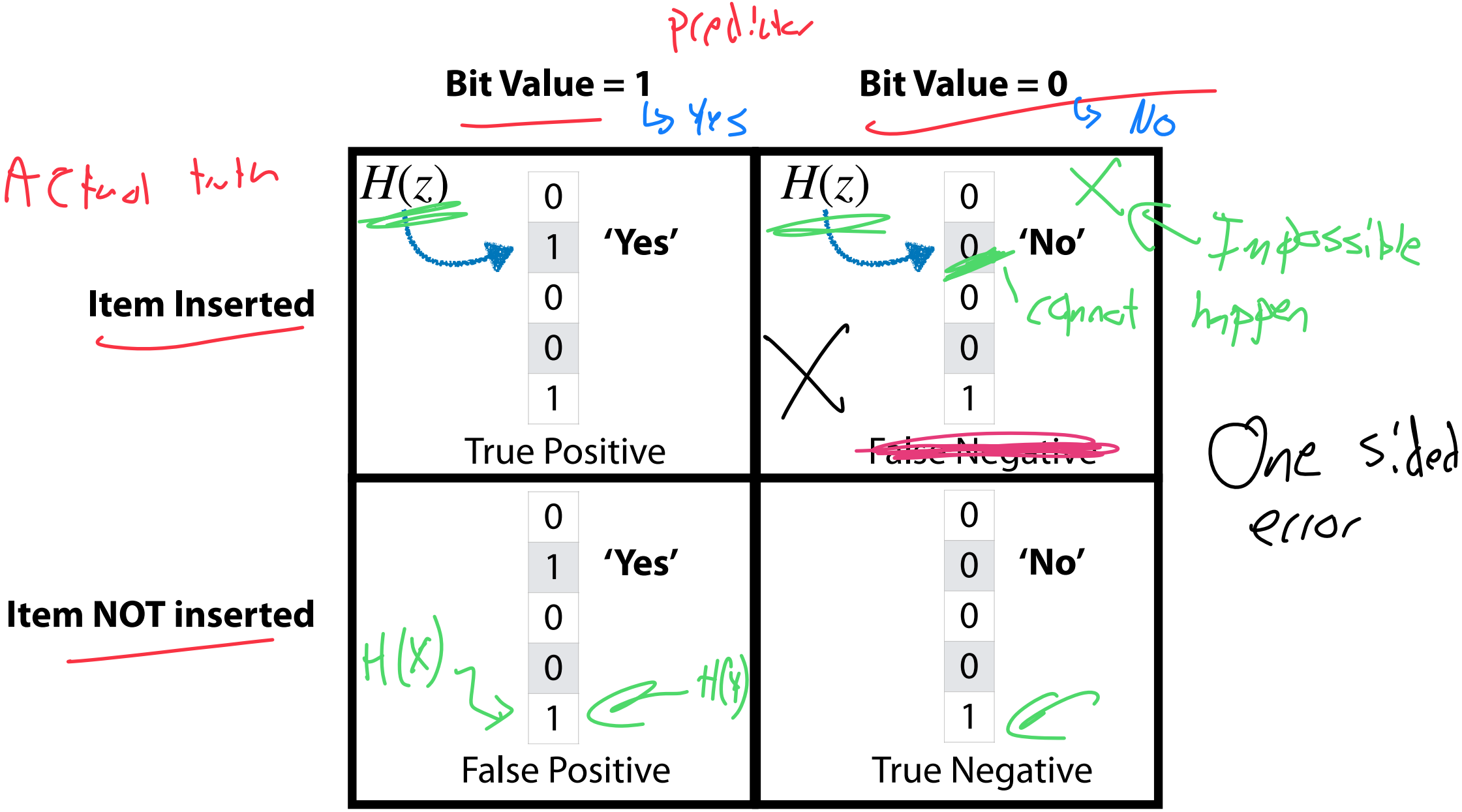
Insert / Find runs in: $O(k) \sim O(1)$

Delete is not possible (yet)!

In real life possible

↳ One sided error b/c we don't delete

Imagine we have a **bloom filter** that **stores malicious sites…**



Bloom Filter: Error Rate

$$FPR = \left(1 - e^{\frac{-nk}{m}}\right)^k$$

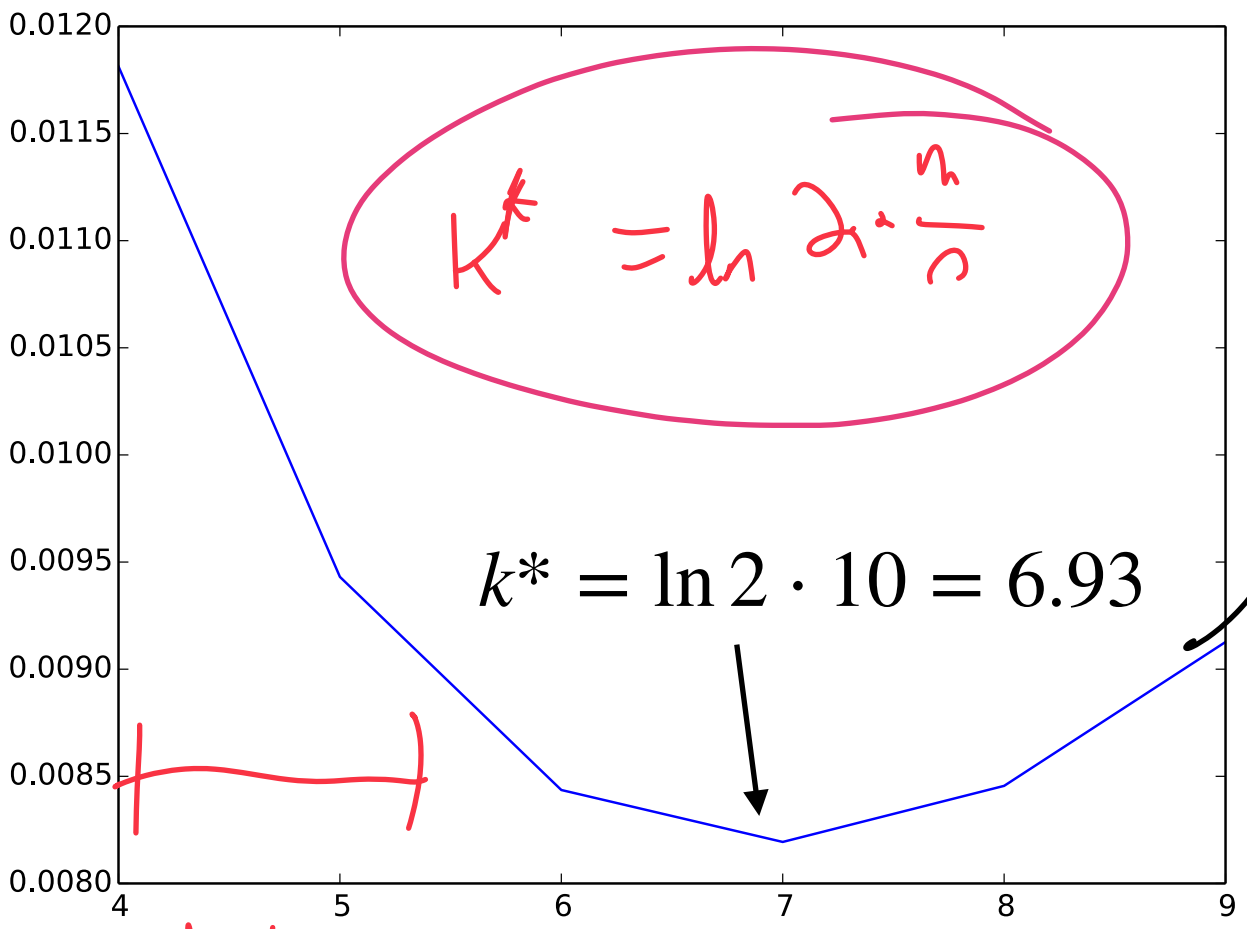


Figure by Ben Langmead

Cardinality Estimation

Let min = 95. Can we estimate N, the cardinality of the set?

Conceptually: If we scatter N points randomly across the interval, we end up with $N + 1$ partitions, each about $1000/(N + 1)$ long

Assuming our first 'partition' is about average:

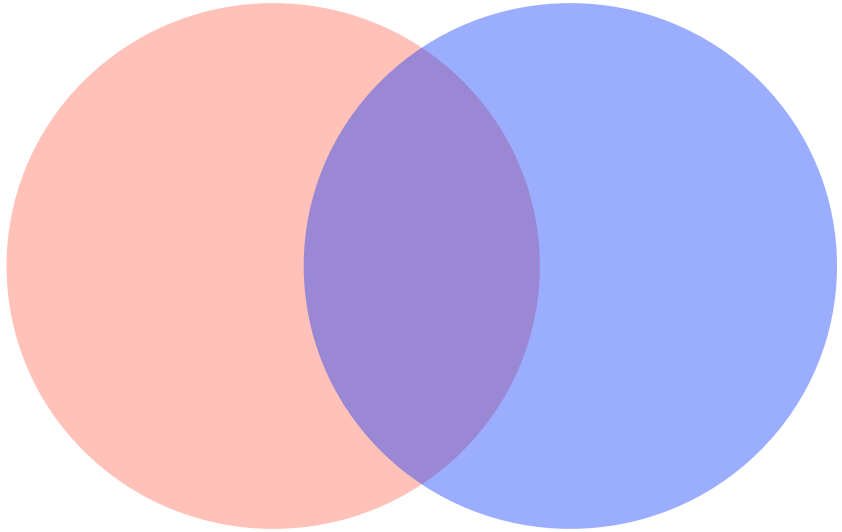
$$95 \approx 1000/(N + 1)$$

$$N + 1 \approx 10.5$$

$$N \approx 9.5$$

Set Similarity Review

To measure **similarity** of A & B, we need both a measure of how similar the sets are but also the total size of both sets.



$$J = \frac{|A \cap B|}{|A \cup B|}$$

J is the ***Jaccard coefficient***

MinHash Sketch

Claim: Under SUHA, set similarity can be estimated by sketch similarity!

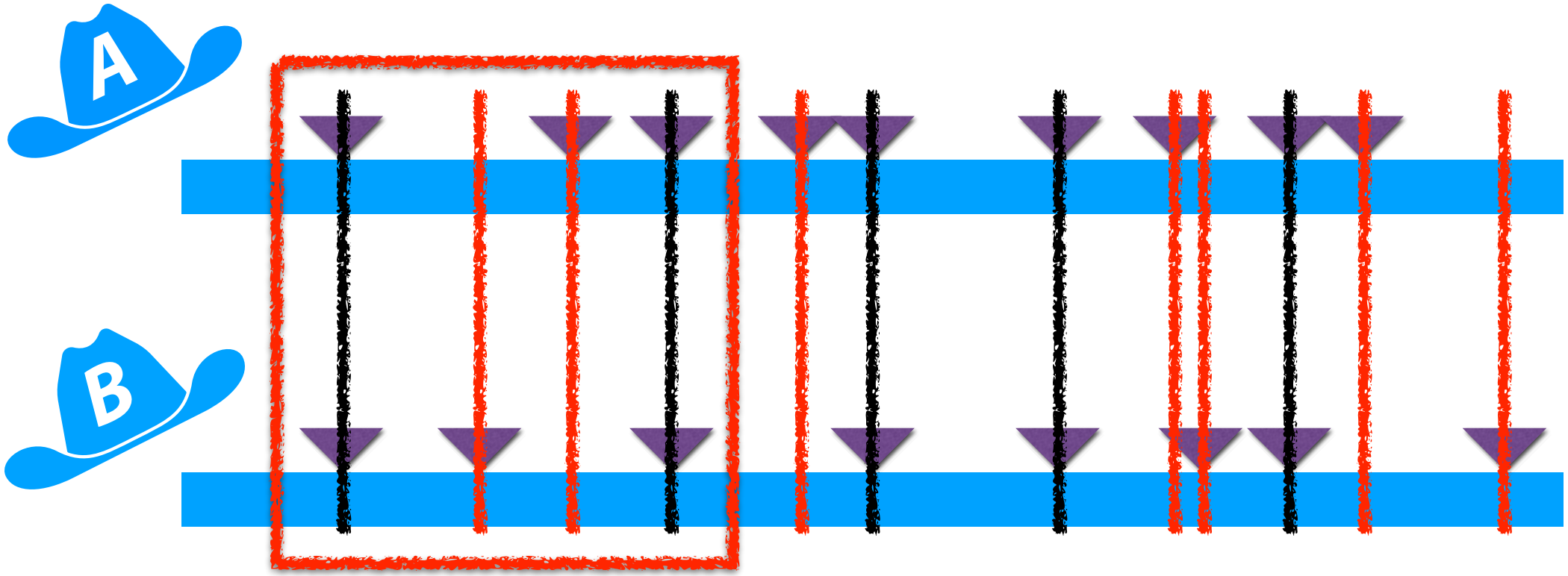
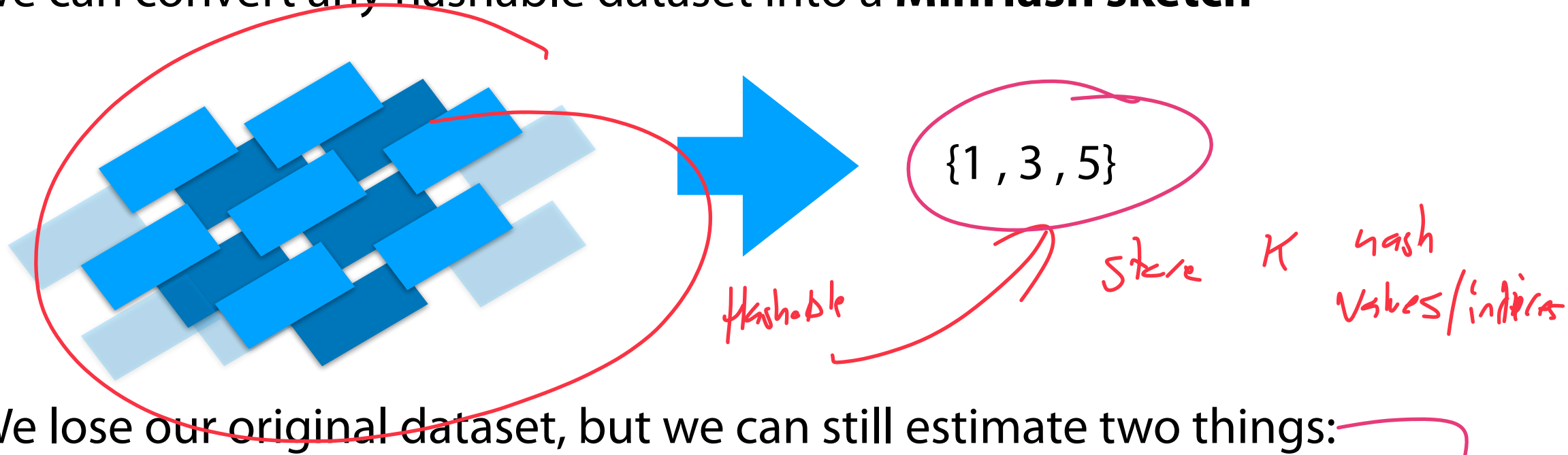


Image inspired by: Ondov B, Starrett G, Sappington A, Kostic A, Koren S, Buck CB, Phillippy AM. **Mash Screen: high-throughput sequence containment estimation for genome discovery**. *Genome Biol* 20, 232 (2019)

MinHash Sketch

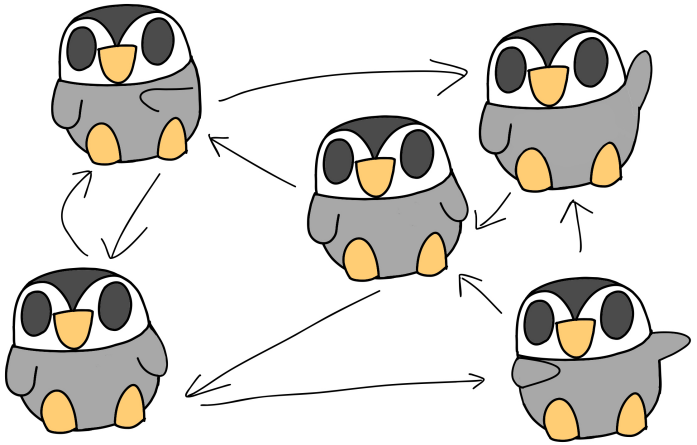
We can convert any hashable dataset into a **MinHash sketch**



We lose our original dataset, but we can still estimate two things:

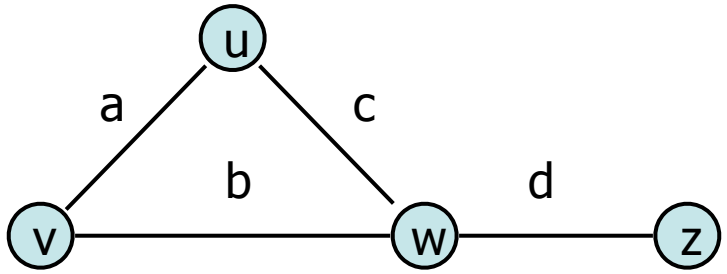
1. Cardinality (k min hash is best estimate)
2. Similarity (Directly or indirectly)

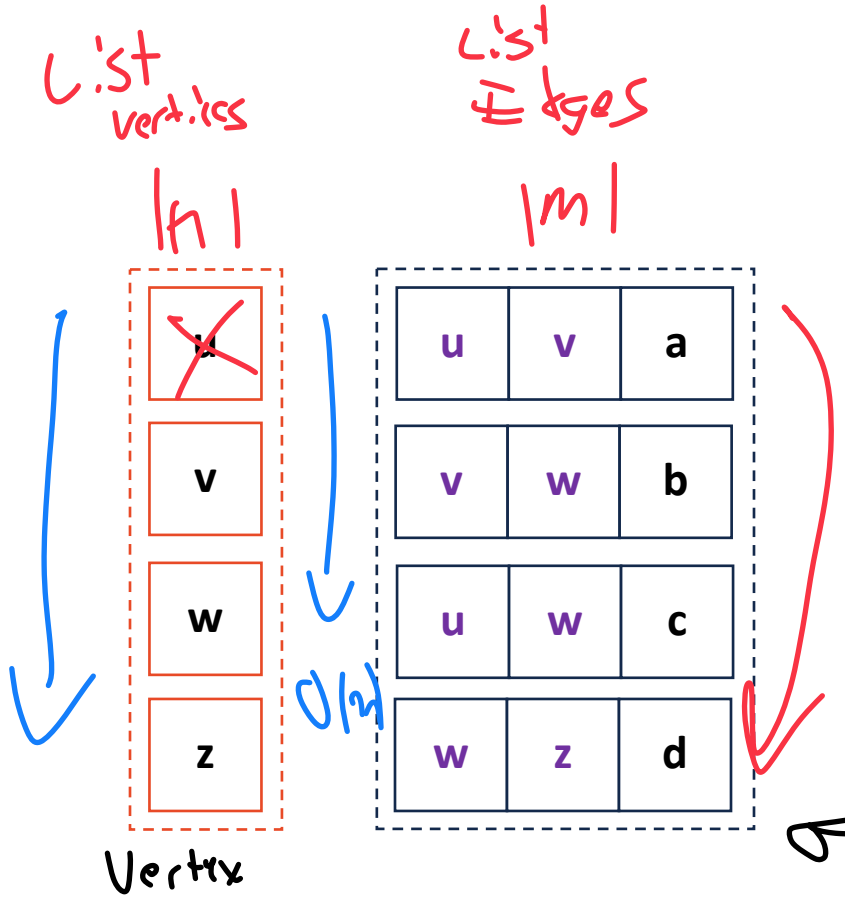
Graphs



November 15 (Graph Implementation 2)

Graph Implementation: Edge List





Big O

insertVertex(K key):

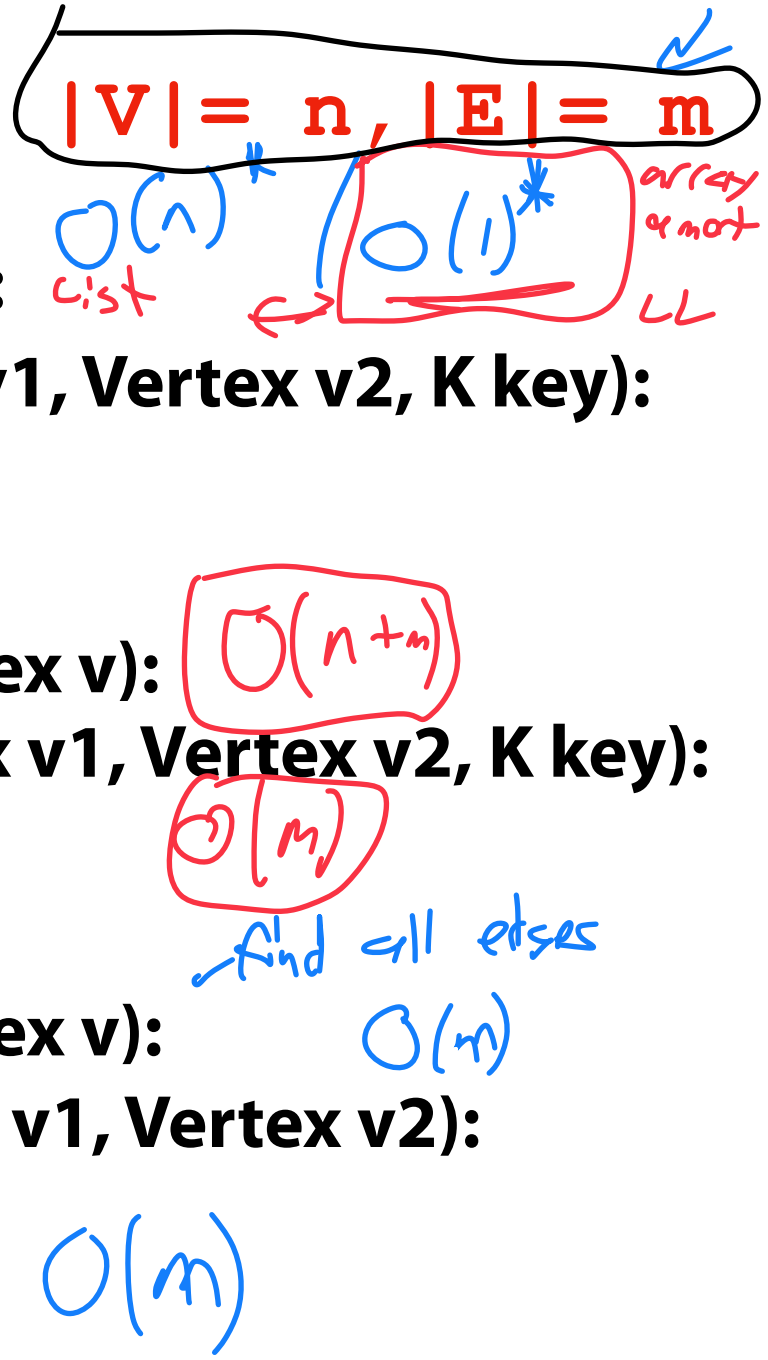
insertEdge(Vertex v1, Vertex v2, K key):

removeVertex(Vertex v):

removeEdge(Vertex v1, Vertex v2, K key):

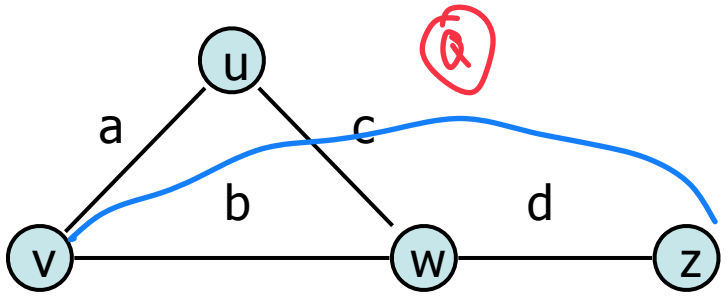
incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):



Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



vertex list H.T.

Edges 2D matrix $O(n^2)$

	u	v	w	z
u	-	a	c	0
v		-	b	0
w			-	d
z				-

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

insertEdge(Vertex v1, Vertex v2, ~~K key~~): $O(1)$ (weight)

removeVertex(Vertex v): $O(n)$

removeEdge(Vertex v1, Vertex v2, ~~K key~~): $O(1)$

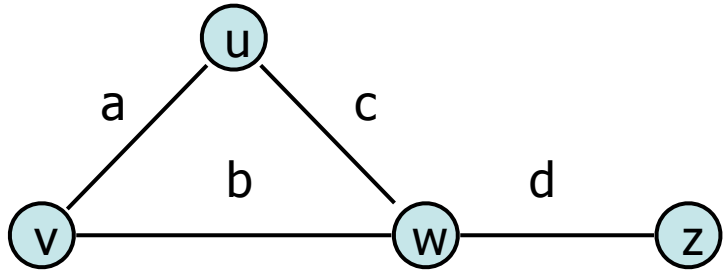
incidentEdges(Vertex v): $O(n)$

areAdjacent(Vertex v1, Vertex v2): $O(1)$

Great for dense graphs!

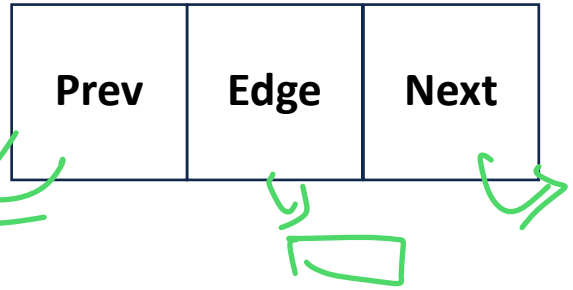
Adjacency List

$|V| = n, |E| = m$

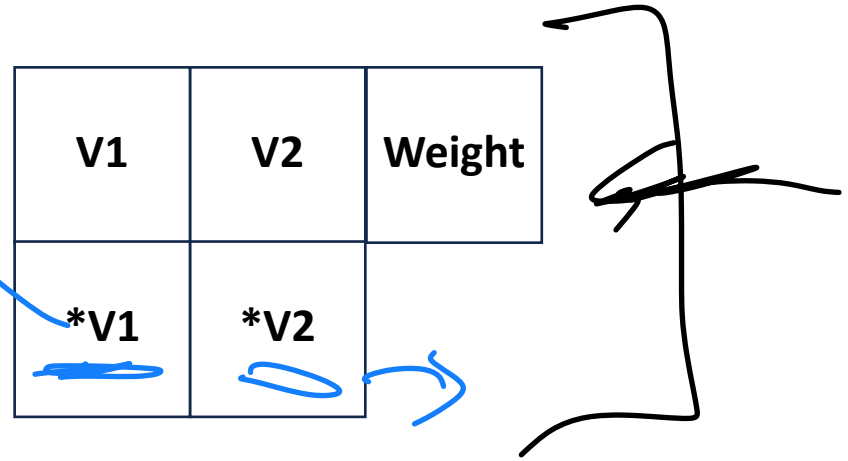


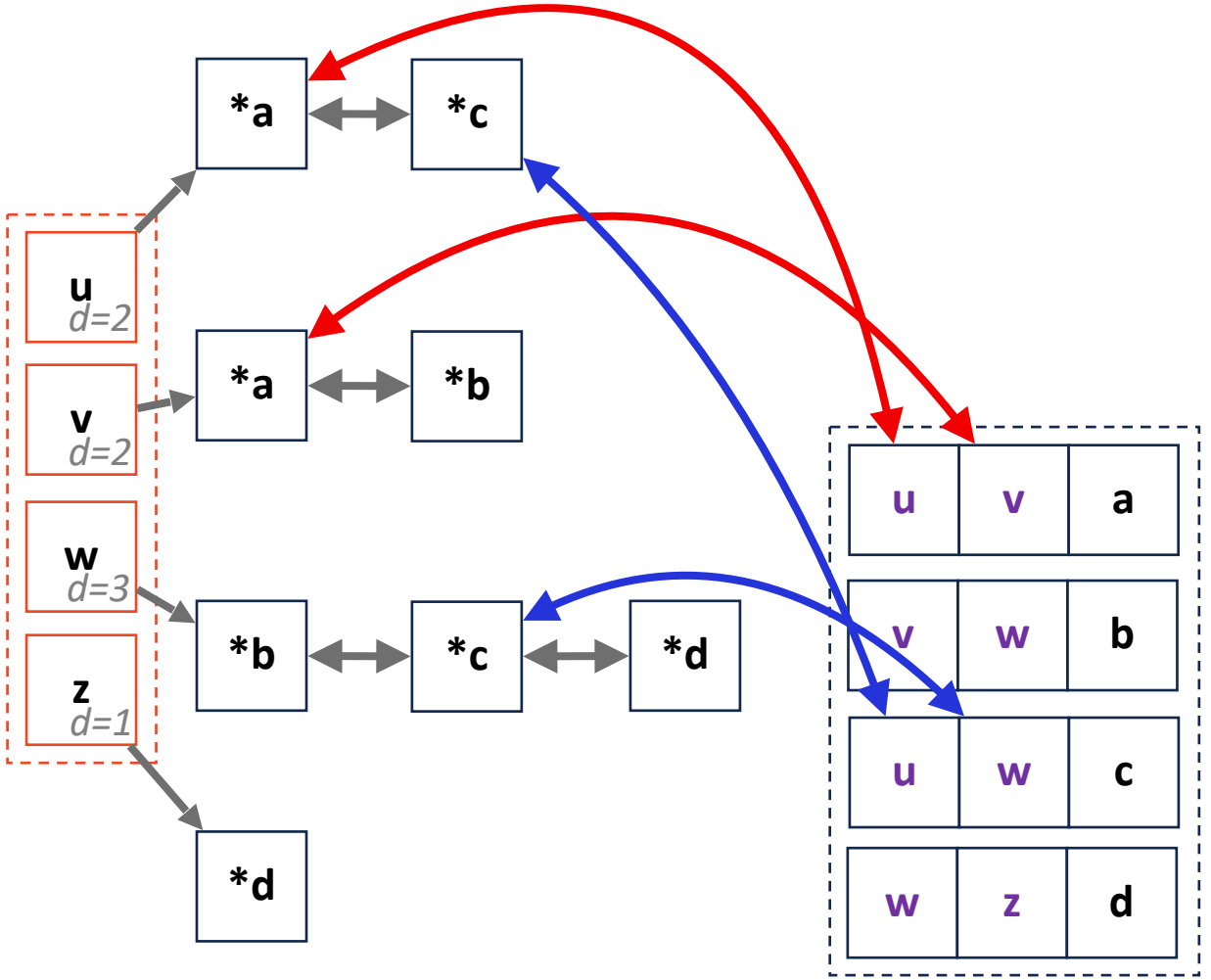
Adj List Node: Bidirectional LL (w/ tail pointer)

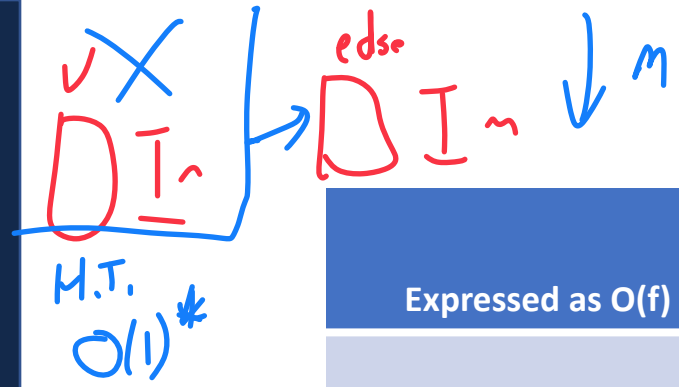
3 pointers



Edge List: Edge list + 2 pointers





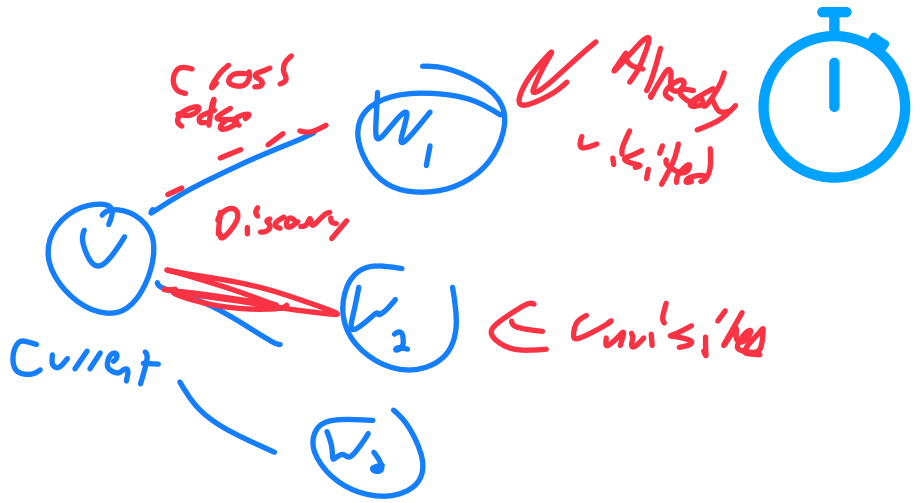


$|V| = n, |E| = m$

Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	m**	n	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
incidentEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

```
1  BFS(G):
2    foreach (Vertex v : G.vertices()):
3      setPred(v, NULL)
4      setDist(v, -1)
5
6    foreach (Edge e : G.edges()):
7      setLabel(e, UNEXPLORED)
8
9    foreach (Vertex v : G.vertices()):
10     if getDist(v) == -1:
11       BFS(G, v)
```

```
12 BFS(G, v):
13   Queue q
14   setDist(v, 0)
15   q.enqueue(v)
16
17   while !q.empty():
18     v = q.dequeue()
19
20     foreach (Vertex w : G.adjacent(v)):
21       if( getDist(w) == -1):
22         setLabel((v, w), DISCOVERY)
23         setPred(w, v)
24         setDist(w, v + 1)
25         q.enqueue(w)
26       else if edge unlabeled
27         setLabel((v, w), CROSS)
```



initialize

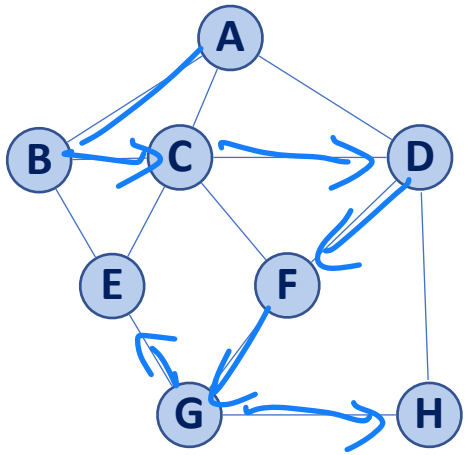
Look at edge

unvisited vertex/edges gets labeled

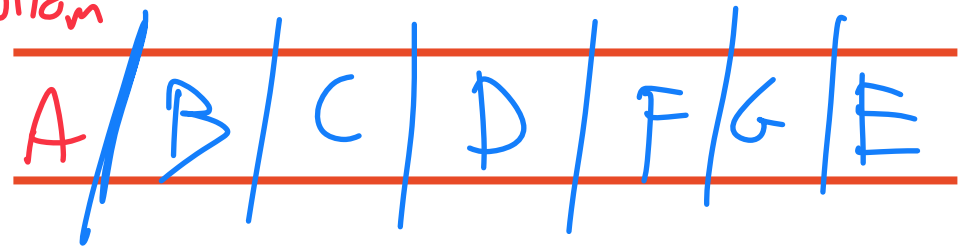
November 17
(Graph Traversal)

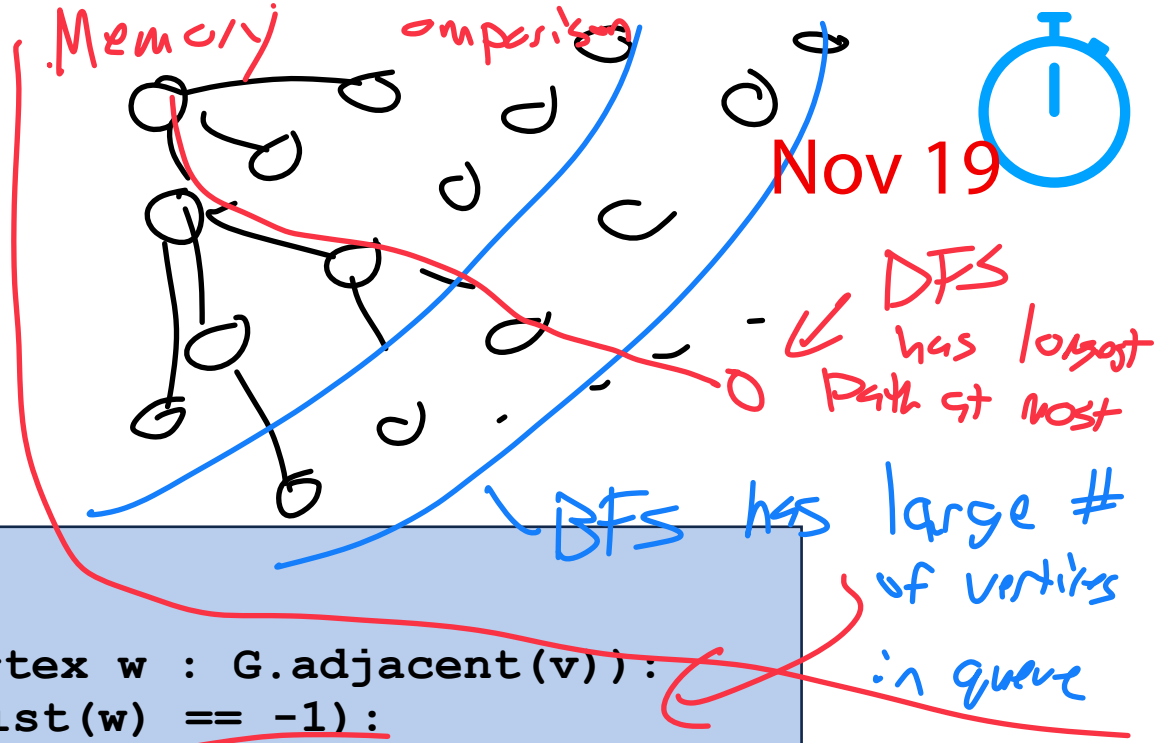
```
1  DFS(G):
2    foreach (Vertex v : G.vertices()):
3      setPred(v, NULL)
4      setDist(v, -1)
5
6    foreach (Edge e : G.edges()):
7      setLabel(e, UNEXPLORED)
8
9    foreach (Vertex v : G.vertices()):
10     if getDist(v) == -1:
11       DFS(G, v)
```

```
12 DFS(G, v):
13
14     foreach (Vertex w : G.adjacent(v)):
15       if( getDist(w) == -1):
16         setLabel((v, w), DISCOVERY)
17         setPred(w, v)
18         setDist(w, v + 1)
19         DFS(G, w)
20       else:
21         setLabel((v, w), BACK)
```



Bottom





A → B, C, D
stop!
↳ DFS(B) → DFS(C) → DFS(D)

Running time of DFS

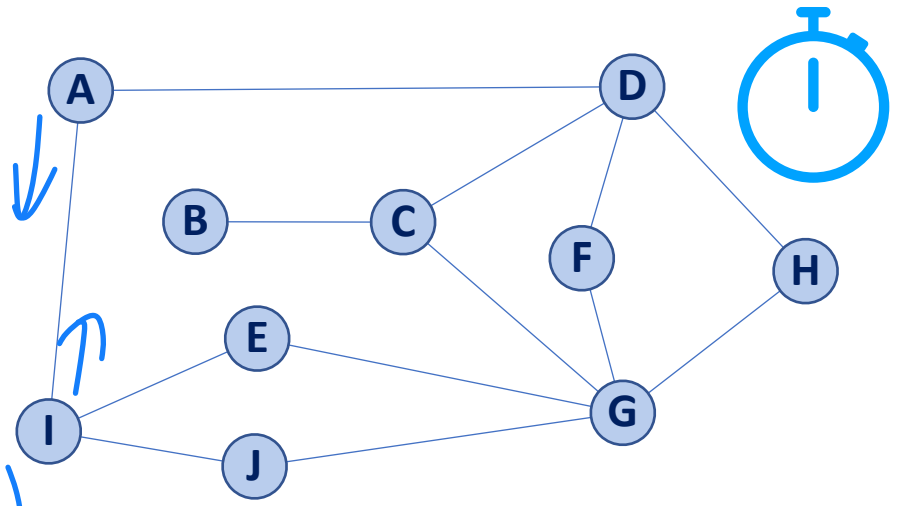
$|V| = n \quad |E| = m$

Labeling:

- Vertex: $O(n)$
- Edge: $O(m)$

BFS: $O(n+m)$

DFS: $O(n+m)$



Traversal:

- Vertex: Each vertex looks at all neighbors
  Look at all vertices
- Edge: $O(m)$

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

$O(m)$

BFS
↳ MST, shortest path, cycles
↳ Memory Hog!

DFS → mem is "bounded" by longest path
↳ MST, cycles
↳ If early termination

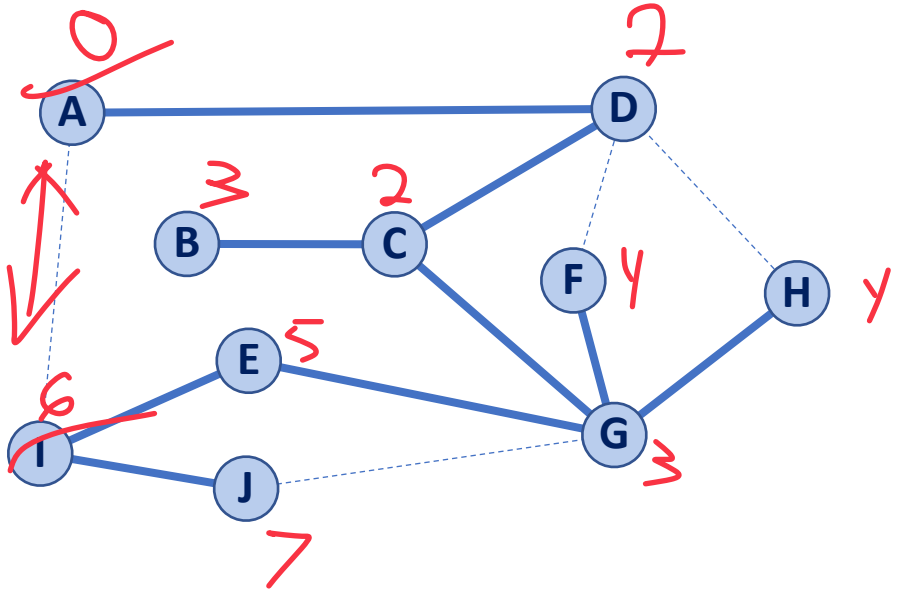
BFS Observations

1. BFS can be used to count components
2. BFS can be used to detect cycles

3. The BFS ‘distance’ value is always the shortest distance from source to any vertex (and the discovery edges form a MST)

4. The endpoints of a cross edge never differ in distance by more than 1.

Traversal: DFS



Do we still make a spanning tree?

↳ Yes it makes a MST*

* on unweighted graphs

Does distance have meaning here?

↳ Has no real meaning!

Discovery Edge

Back Edge

Do our edge labels have meaning here?

↳ Back edge still implies cycle

Not cross b/c cross implies dist(u,v) ∈ 1

Kruskal's Algorithm

Priority Queue:	Total Running Time
Heap	O(n + m + m log(n)) *
Sorted Array	O(n + m log(n) + m)

Construction

get / remove Min

1) Disjoint Set + Priority Queue

* Destructive

Sorted array persist

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

O(n)

Prim's Algorithm

Sparse Graph: $O(n) \leq O(m) \leq O(n^2)$

↳ $m \sim n$ → $O(n \log n)$

↳ adj. list heap is best

Dense Graph: $m \sim n^2$

↳ unsorted array is better than heap on dense graph

```
 6  PrimMST(G, s):
 7    foreach (Vertex v : G.vertices()):
 8      d[v] = +inf
 9      p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T          // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

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

$n^2 \log n$

Dijkstra's Algorithm (SSSP)

What is the running time of Dijkstra's Algorithm?

Running time for Prim

using Fib heap

$O(n + n \log n)$

$n \log n$ or n

```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 7      d[v] = +inf
 8      p[v] = NULL
 9    d[s] = 0
10
11    PriorityQueue Q // min distance, defined by d[v]
12    Q.buildHeap(G.vertices())
13    Graph T          // "labeled set"
14
15    repeat n times:
16      Vertex u = Q.removeMin()
17      T.add(u)
18      foreach (Vertex v : neighbors of u not in T):
19        if cost(u, v) + d[u] < d[v]:
20          d[v] = cost(u, v) + d[u]
21          p[v] = m
22
23    return T
```

O(n)

O(n)

O(log n) -> O(n)

total of m edges

min heap	Fib heap
$O(\log n)$	$O(\log n)$
$O(\log n)$	$O(1)$

removing min

updating / decreasing key

$O(1)$ Fib heap

$O(\log n)$

Dijkstra's Algorithm (SSSP)

Dijkstras Algorithm works only on non-negative weights* efficiently

* no neg cycles allowed

Optimal implementation:

Fib heap

on dense graph ties

unsorted list

Optimal runtime:

$O(m + n \log n)$

$O(n^2)$

```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 7      d[v] = +inf
 8      p[v] = NULL
 9    d[s] = 0
10
11    PriorityQueue Q // min distance, defined by d[v]
12    Q.buildHeap(G.vertices())
13    Graph T          // "labeled set"
14
15    repeat n times:
16      Vertex u = Q.removeMin()
17      T.add(u)
18      foreach (Vertex v : neighbors of u not in T):
19        if cost(u, v) + d[u] < d[v]:
20          d[v] = cost(u, v) + d[u]
21          p[v] = m
22
23    return T
```

Floyd-Warshall Algorithm

Running time?

$O(n^3)$, easy to code!

→ easy to multithread

↳ textbook dynamic program

```
     FloydWarshall(G):
 6     Let d be a adj. matrix initialized to +inf
 7     foreach (Vertex v : G):
 8       d[v][v] = 0
 9     foreach (Edge (u, v) : G):
10       d[u][v] = cost(u, v)
11
12     foreach (Vertex w : G):
13       foreach (Vertex u : G):
14         foreach (Vertex v : G):
15           if (d[u, v] > d[u, w] + d[w, v])
16             d[u, v] = d[u, w] + d[w, v]
```

$O(n)$

$O(m)$

$O(n^3)$

$n \times$

$n \times$

$n \times$

$O(1)$