Data Structures
Review

CS 225
Brad Solomon

December 6, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Represents only an attempt to provide some helpful resources.

Requested: Iterators

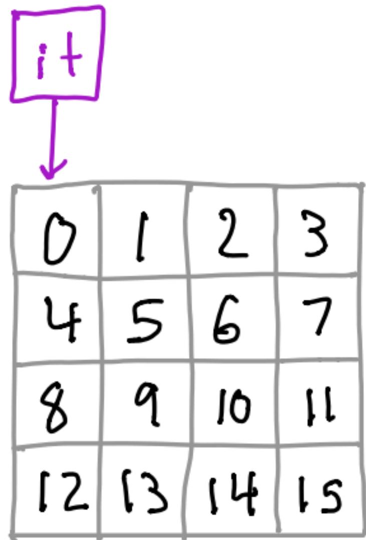
```
1
2  std::vector<Animal> zoo;
3
4
5  /* Full text snippet */
6
7    for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
8      std::cout << (*it).name << " " << (*it).food << std::endl;
9    }
10
11
12 /* Auto Snippet */
13
14   for ( auto it = zoo.begin(); it != zoo.end; ++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
```

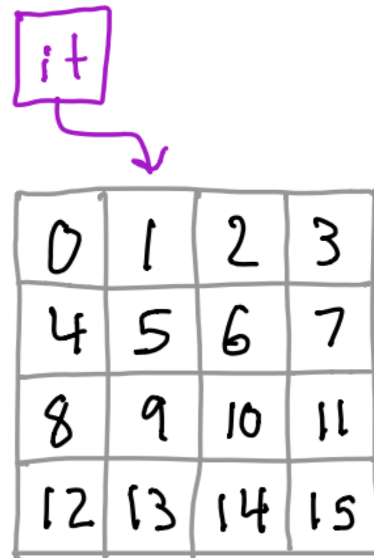
Iterators

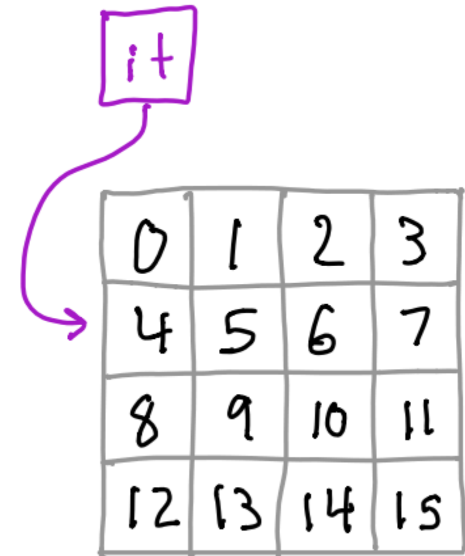
Iterators defined inside a class (and as part of class)

```
 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)

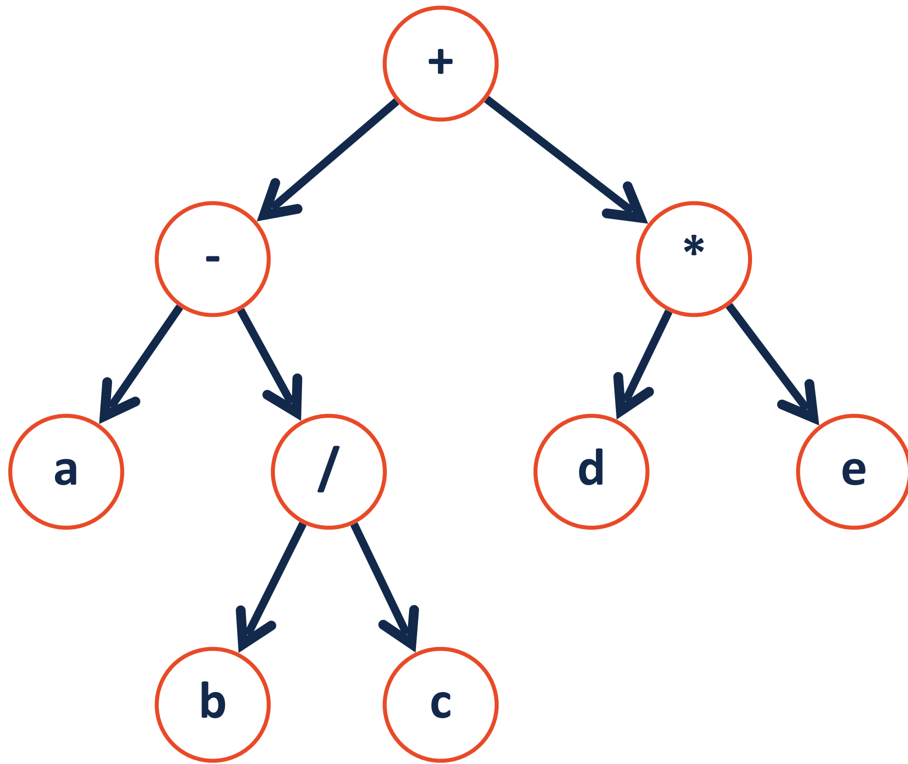






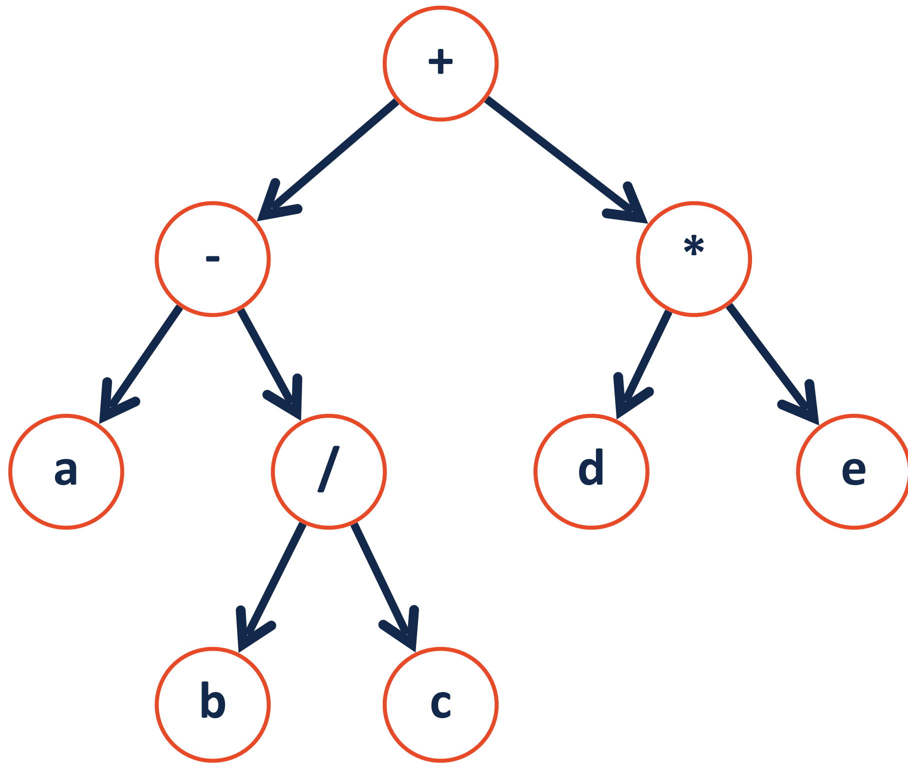
Requested: Tree Traversals

Traversals



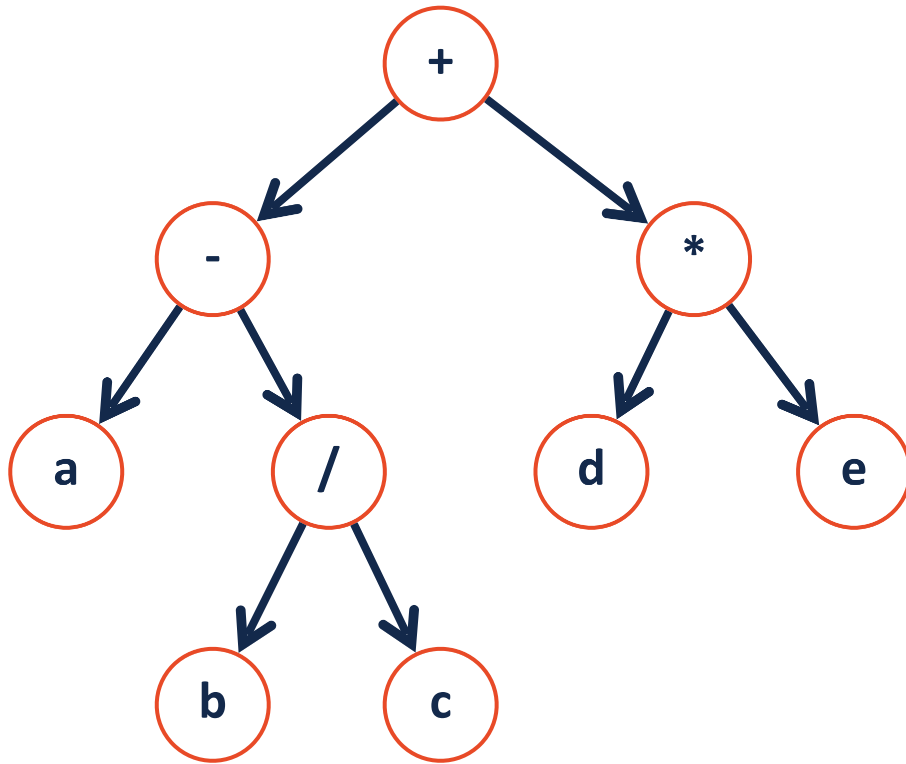

```
 1  template<class T>
 2  void BinaryTree<T>::_____Order(TreeNode * root)
 3  {
 4
 5     if (root) {
 6
 7           ___________________________;
 8
 9           _____Order(root->left);
10
11           ___________________________;
12
13           _____Order(root->right);
14
15           ___________________________;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

Traversals



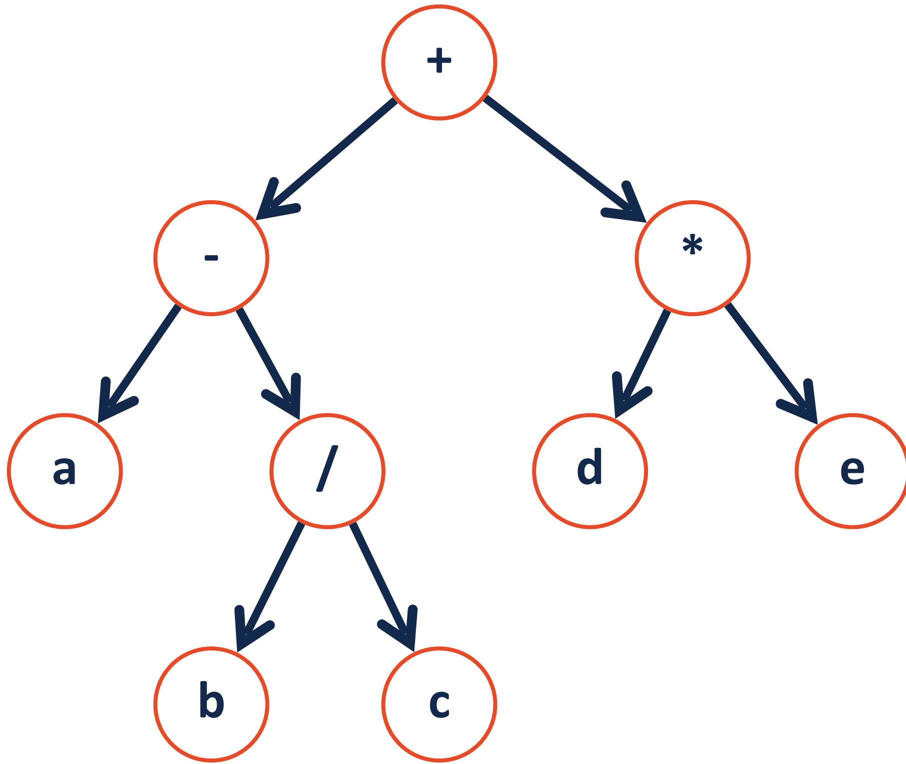

```
1  template<class T>
2  void BinaryTree<T>::_____Order(TreeNode * root)
3  {
4
5    if (root) {
6
7          ______________________________;
8
9          _____Order(root->left);
10
11         ______________________________;
12
13         _____Order(root->right);
14
15         ______________________________;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

Traversals




```
 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  }
```

Level-Order Traversal




```
 1  template<class T>
 2  void BinaryTree<T>::lOrder(TreeNode * root)
 3  {
 4
 5      Queue<TreeNode*> q;
 6      q.enqueue(root);
 7
 8      while( q.empty() == False){
 9
10          TreeNode* temp = q.head();
11          process(temp);
12
13          q.dequeue();
14
15          q.enqueue(temp->left);
16          q.enqueue(temp->right);
17
18      }
19  }
```

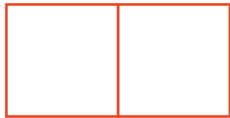
Requested: Amortized Analysis

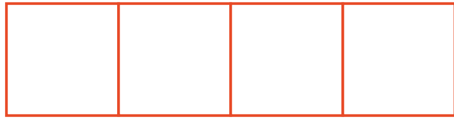
When an algorithm has an infrequent 'costly' step.

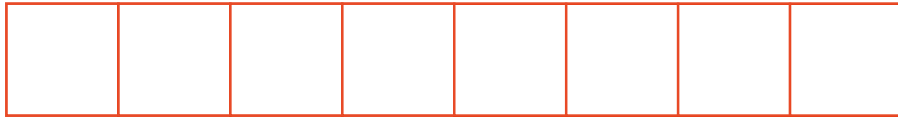
What algorithms *should* we consider from an amortized point of view?

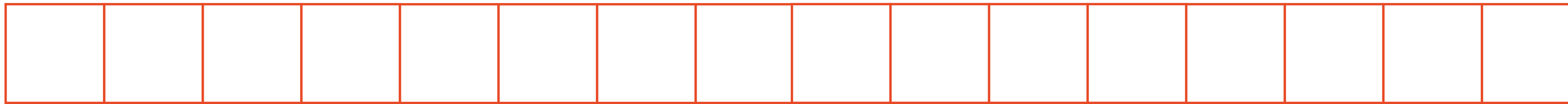
Resize Strategy: x2 elements every time

Total copies for n inserts: 2n -1





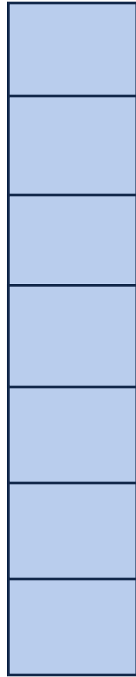




Resize Strategy: x2 elements every time

Resizing a hash table

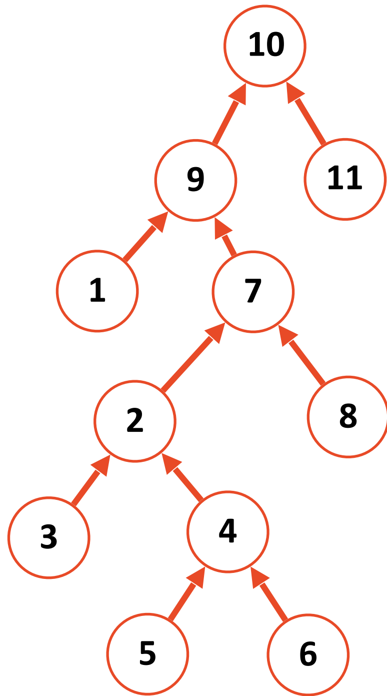
How do you resize?

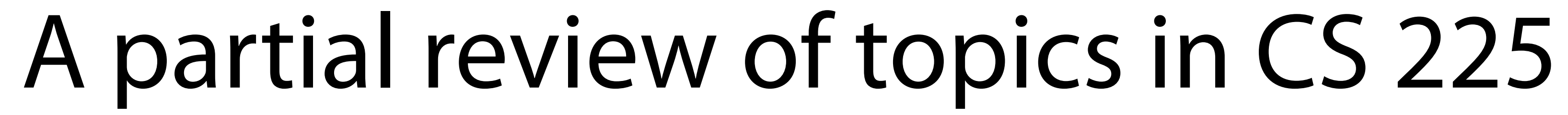


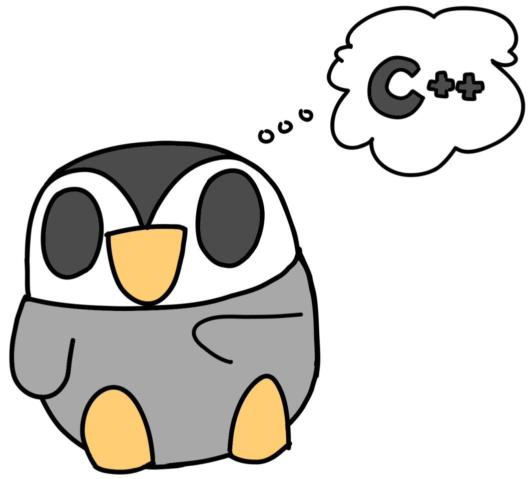
Amortized Time (Rank w/ Path Compression)

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

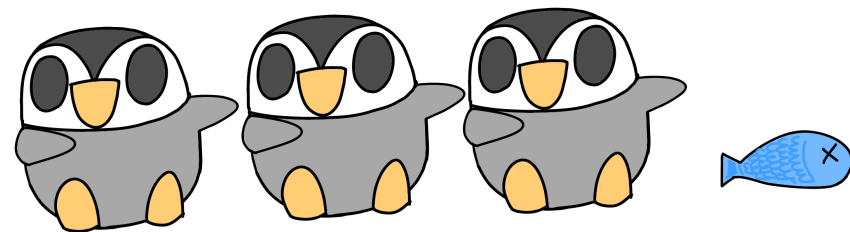
We are interested in the **worst case work** possible **over m calls**.



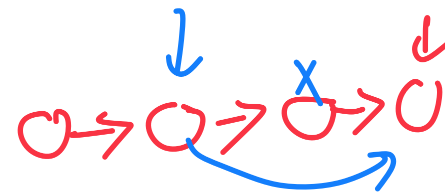

A partial review of topics in CS 225

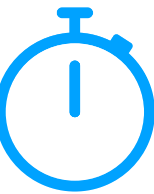

C++

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 (*&)	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)

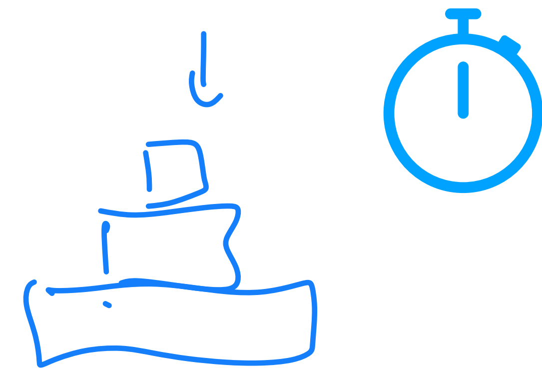
September 6 (Array 2 Lecture)

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)

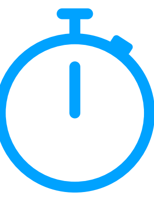
Stack ADT

- [Order]: Last in first out (LIFO)
- [Implementation]: vector / deque ← C++ (LL or array list)
- [Runtime]:
  push() O(1)
  pop() O(1)



September 8 (Quacks Lecture)

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:

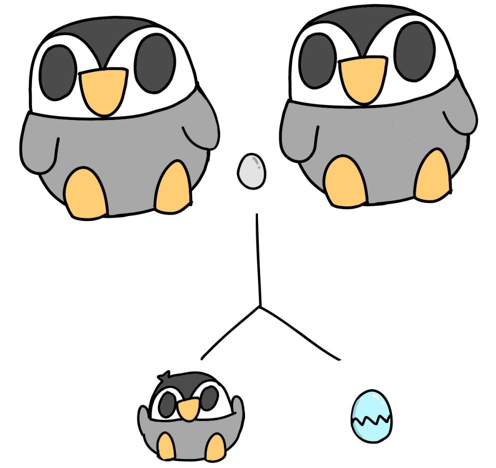
1. It must be of base class **`std::iterator`**

2. It must implement at least the following operations:

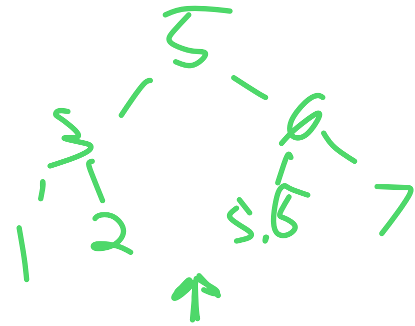
```
Iterator& operator ++()

const T & operator *()

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

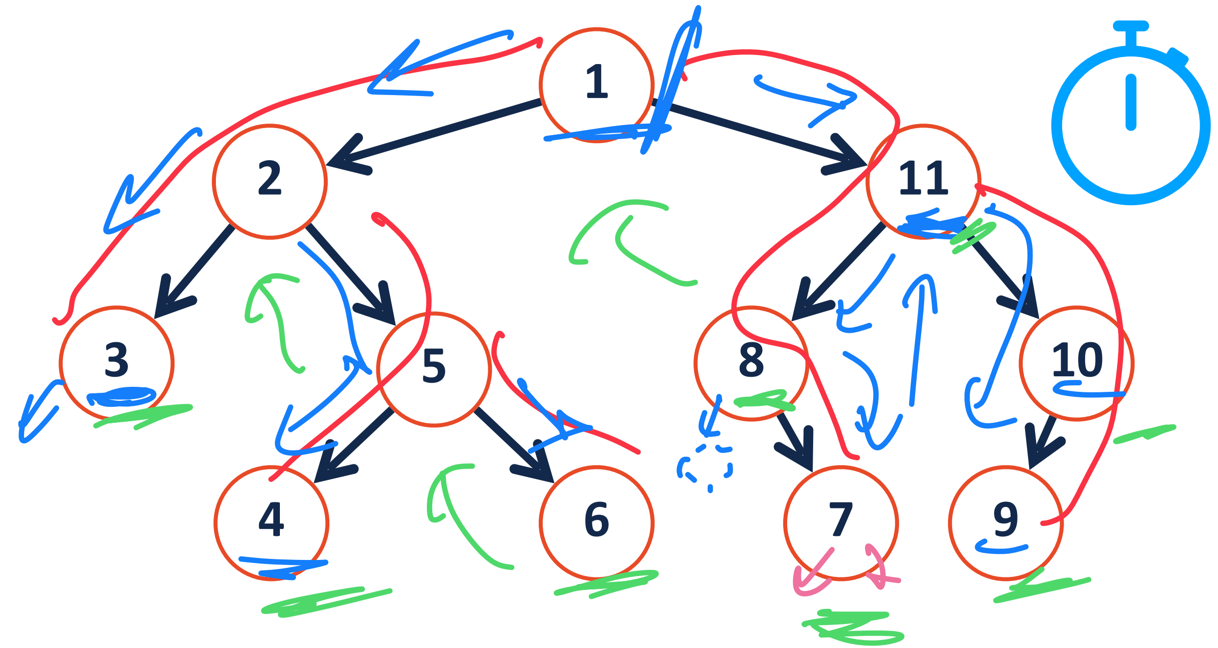
```
1
2   std::vector<Animal> zoo;
3
4
5   /* Full text snippet */
6
7     for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
8       std::cout << (*it).name << " " << (*it).food << std::endl;
9     }
10
11
12  /* Auto Snippet */
13
14    for ( auto it = zoo.begin(); it != 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
```

Trees



Tree Traversals



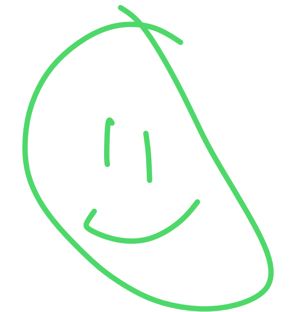


Pre-order: 1 2 3 5 4 6 11 8 7 10 9

7 8 11?

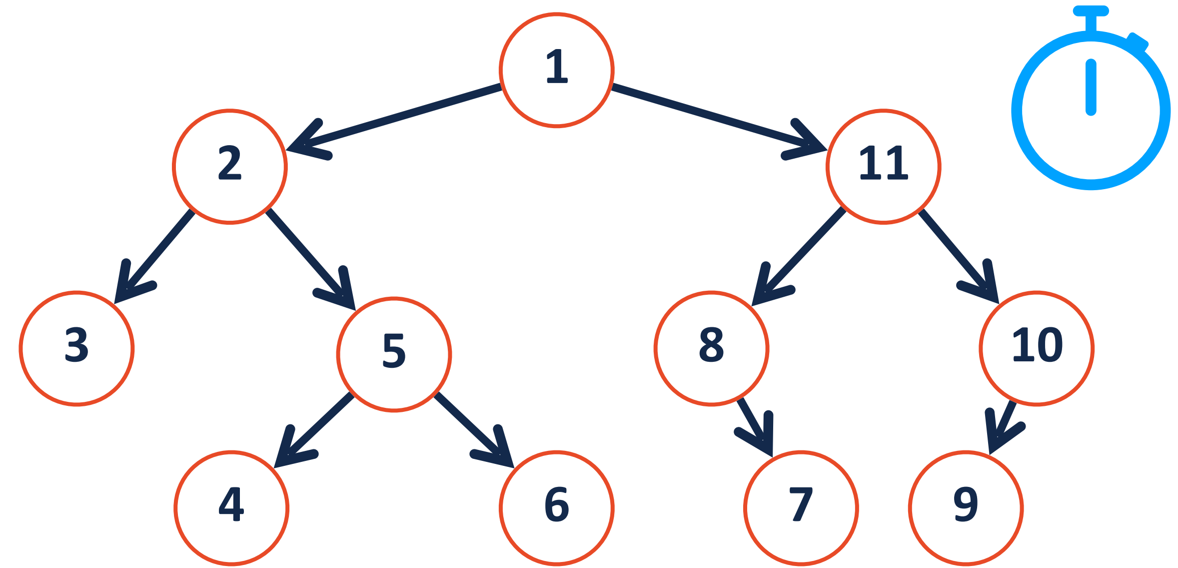
In-order: 3 2 4 5 6 1 8 7 11 9 10

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

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

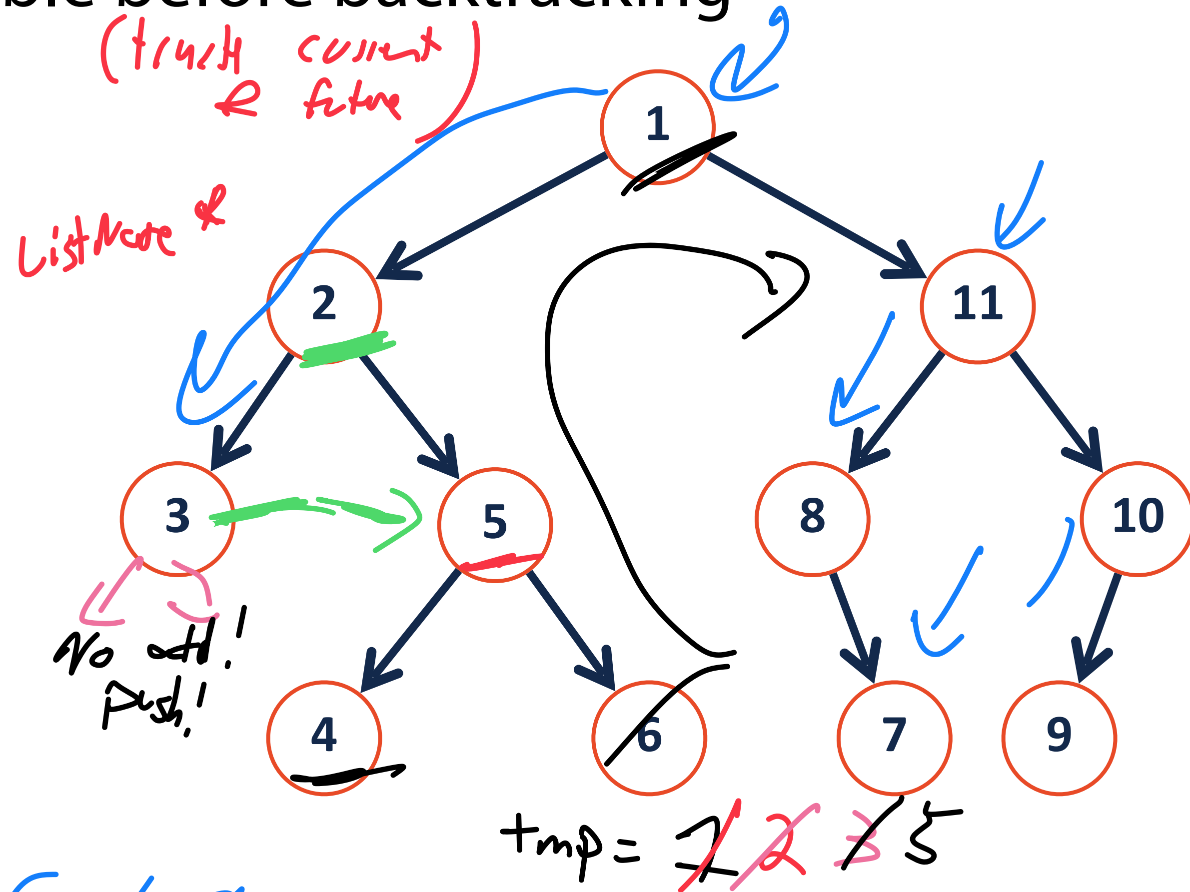
Post-order: 3, 4, 6, 5, 2, 7, 8, 9, 10, 11, 1

September 15 (Tree Traversal)

Depth First Search

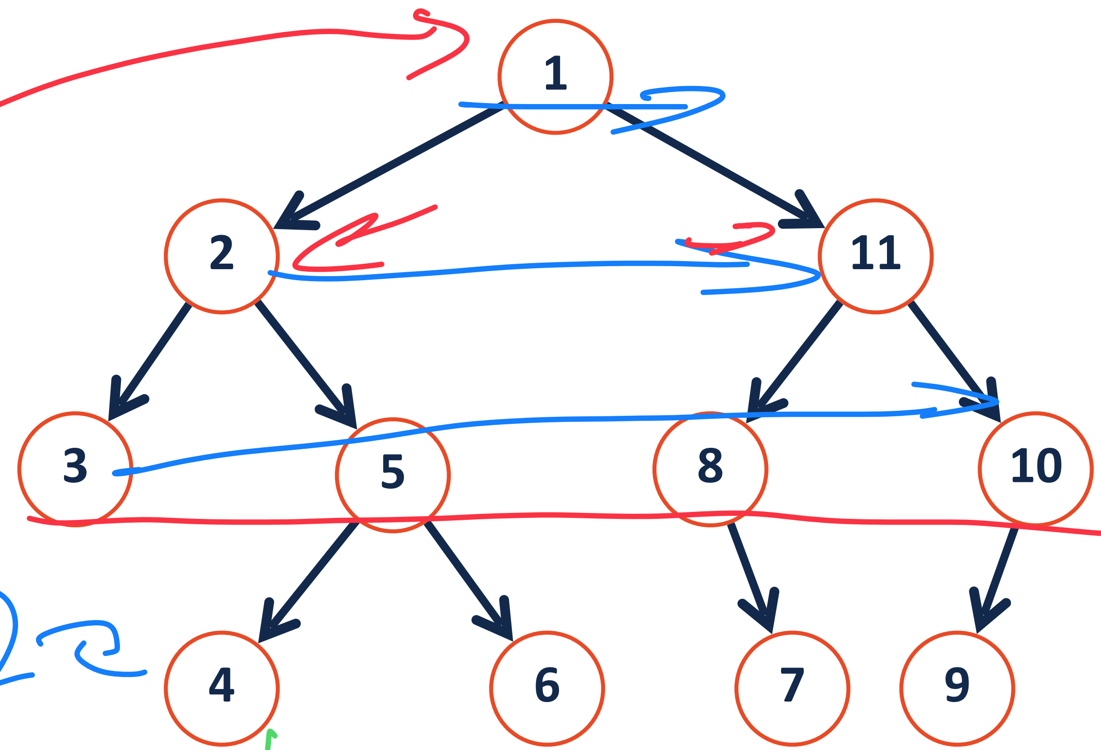
September 18 (BST Lecture)

Explore as far along one path as possible before backtracking



Breadth First Search

Fully explore depth i before exploring depth i+1



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

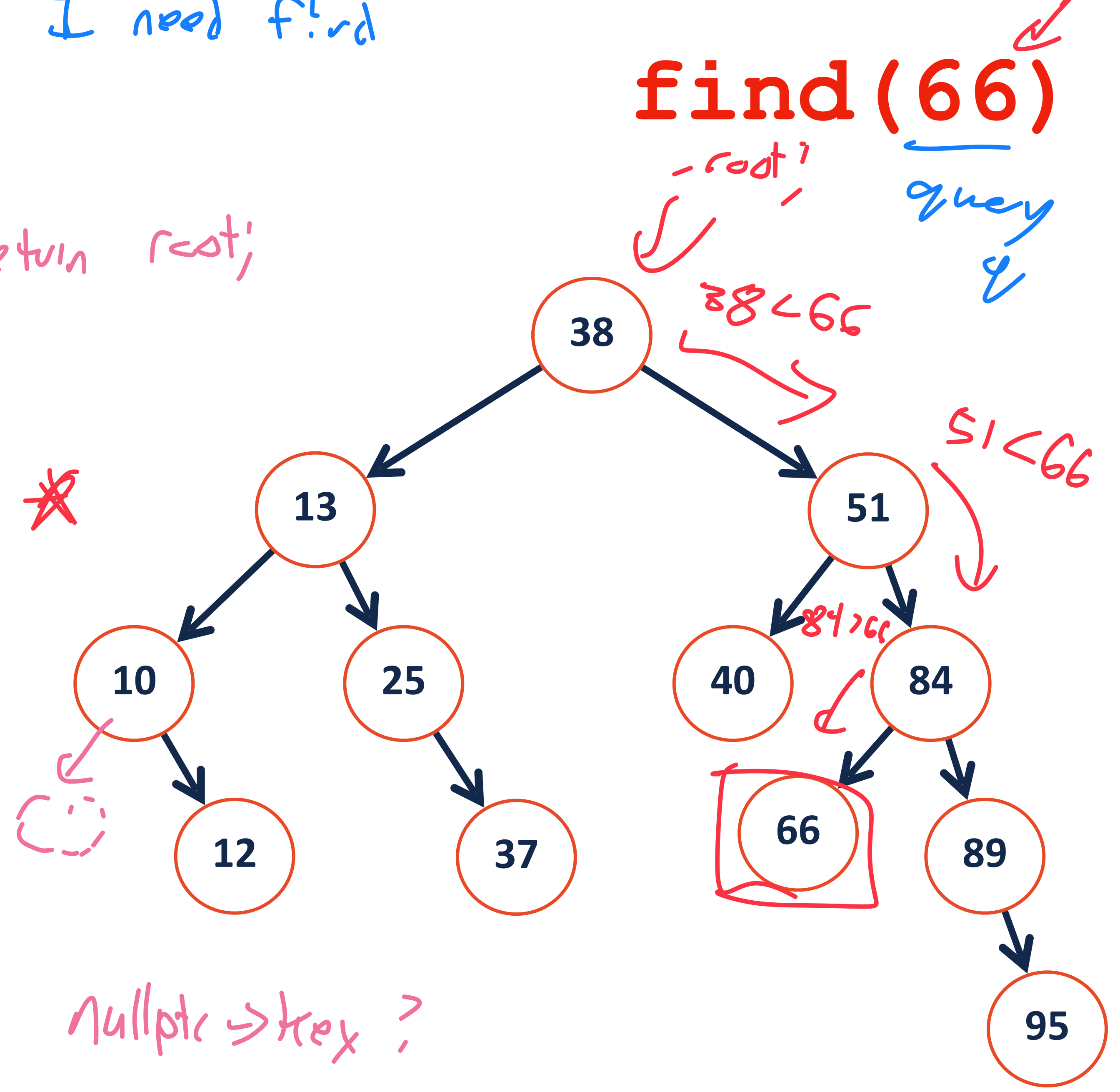
if yes, return root

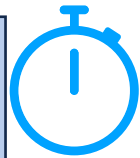
if root->key < query

recurse right

if root->key > query

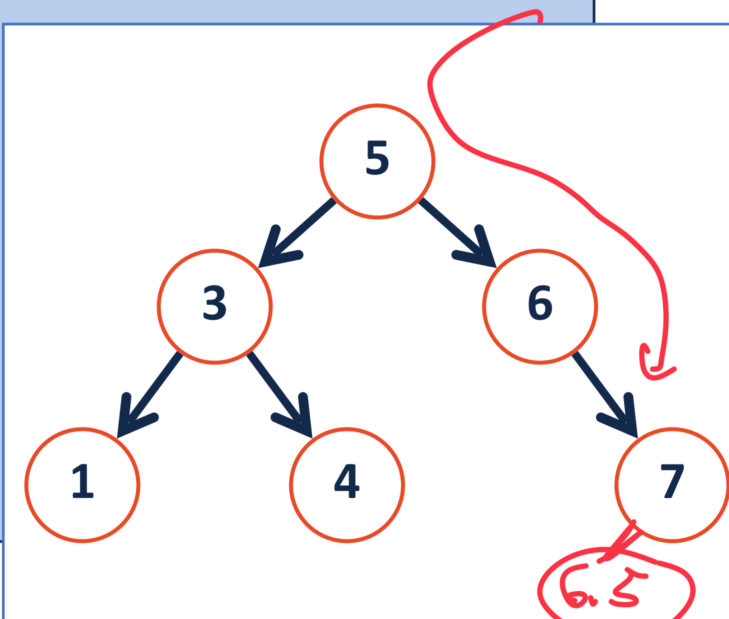
recurse left





```
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
6
7
8
9
10
11
12
13
14
15
16 }
```



September 20 (BST 2)

September 22 (AVL)

```
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

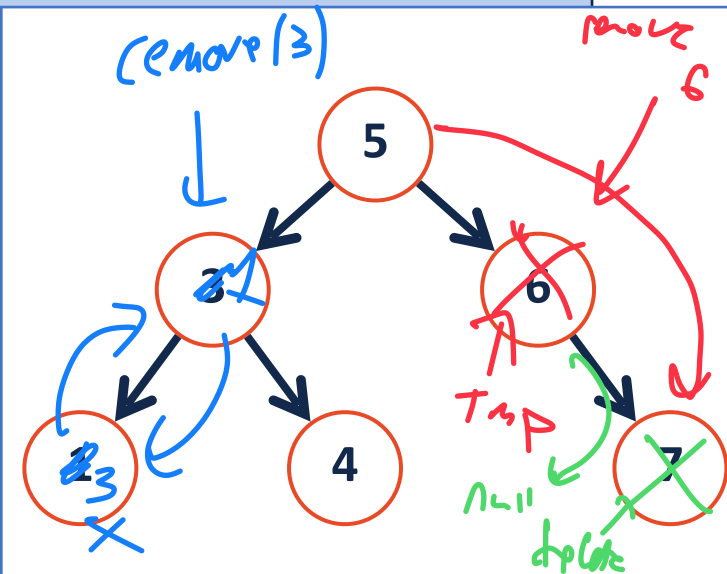
↳ LL remove

2 - child removal

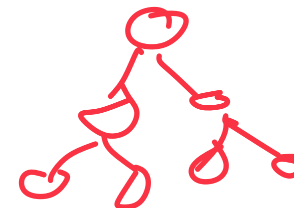
↳ find IOP / IOS

↳ swap IOP w/ root

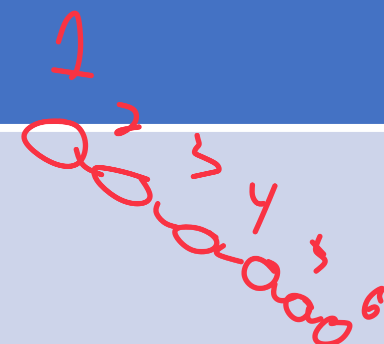
↳ remove (key)



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	$O(n)$



September 20 (BST 2)

BST Analysis – Running Time

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

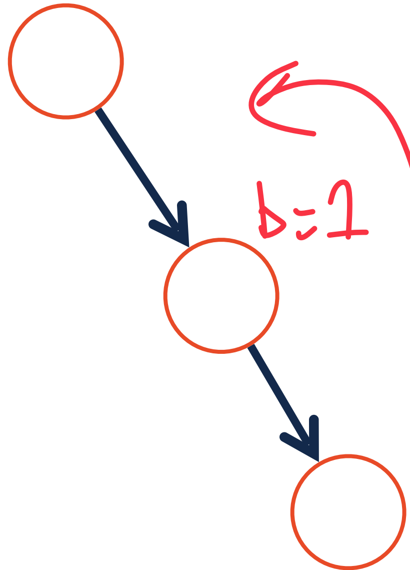
September 20 (BST 2)

AVL Rotations

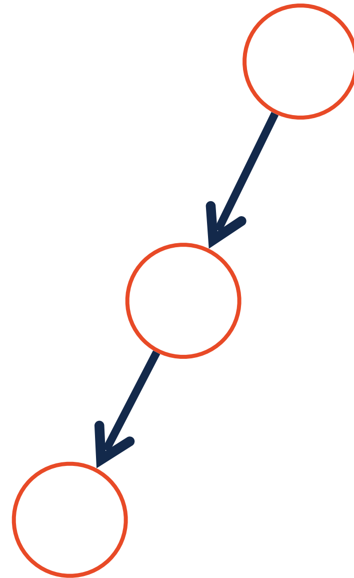
Very important!

Complex

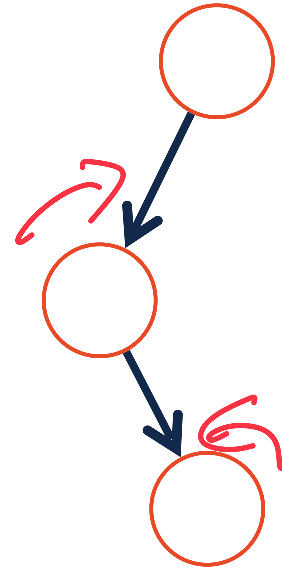
b: 2 Simple



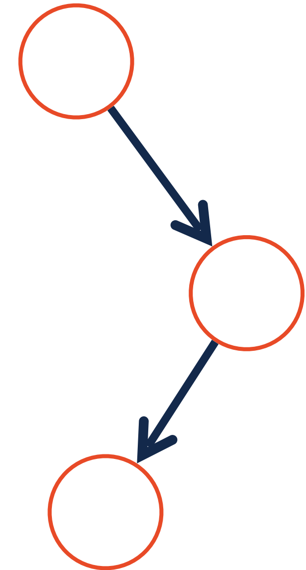
Left



Right



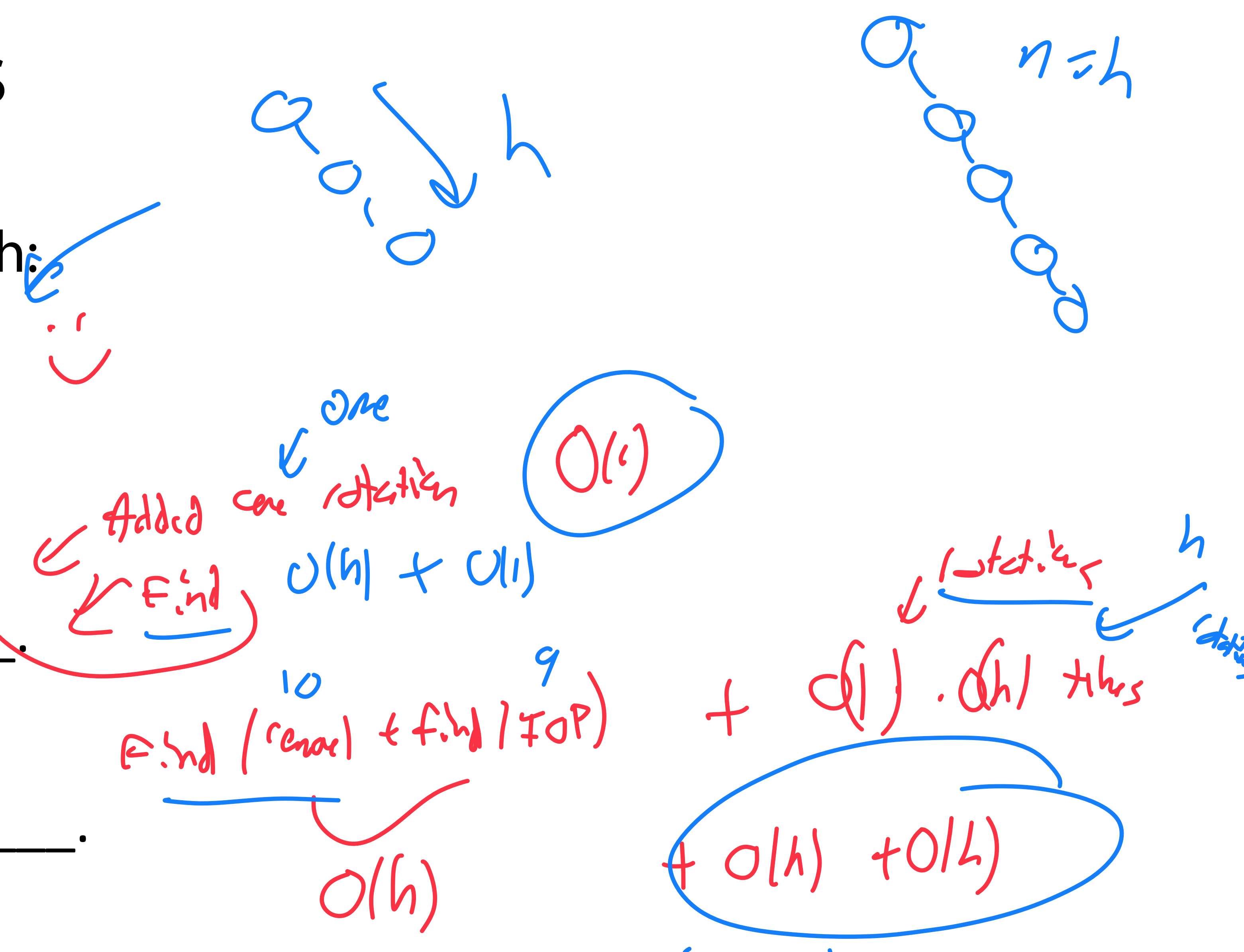
Left Right



Right Left

September 25 (AVL 2)

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)____.

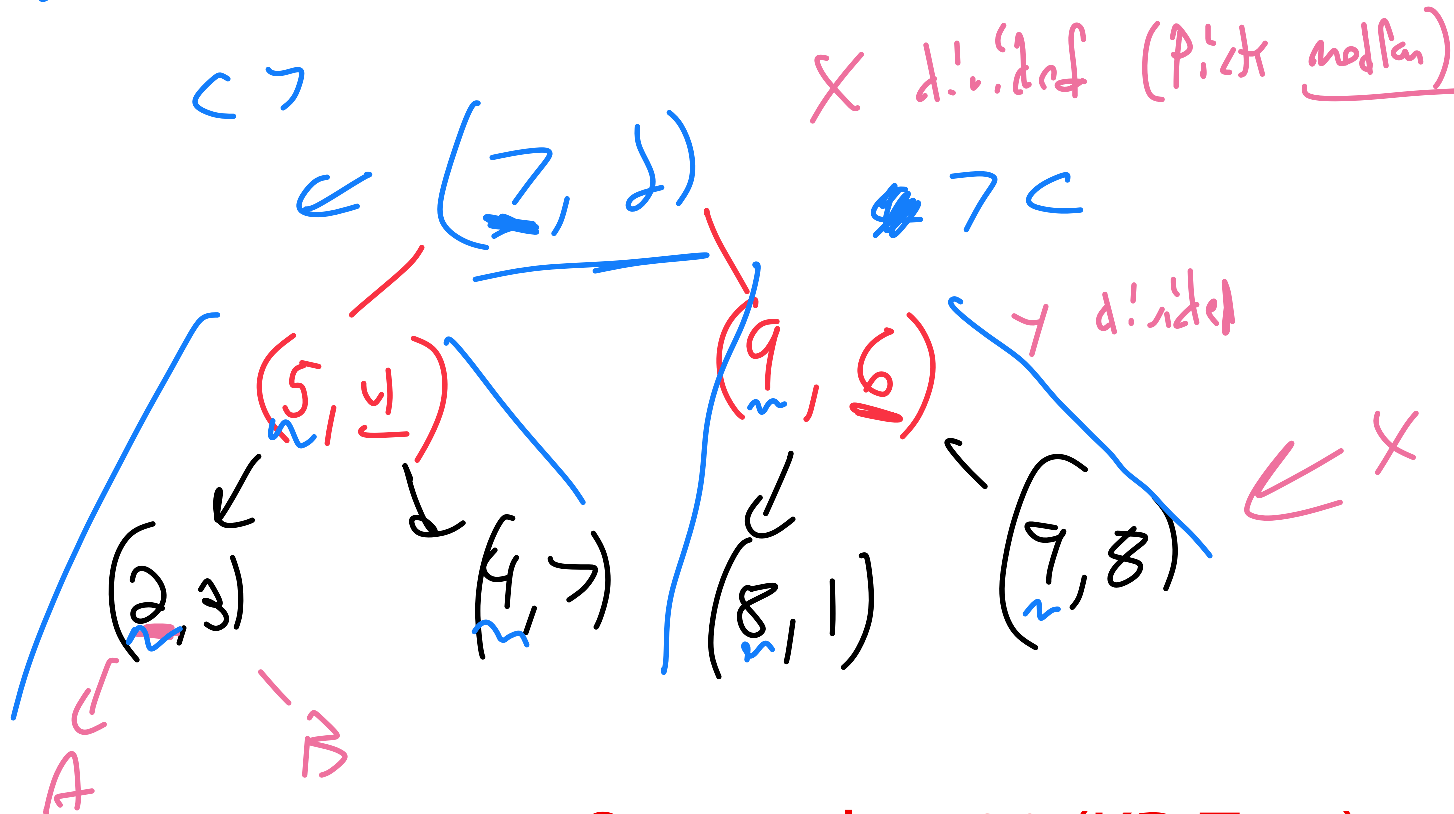
Claim: The height of the AVL tree with n nodes is: ____O(log n)____.

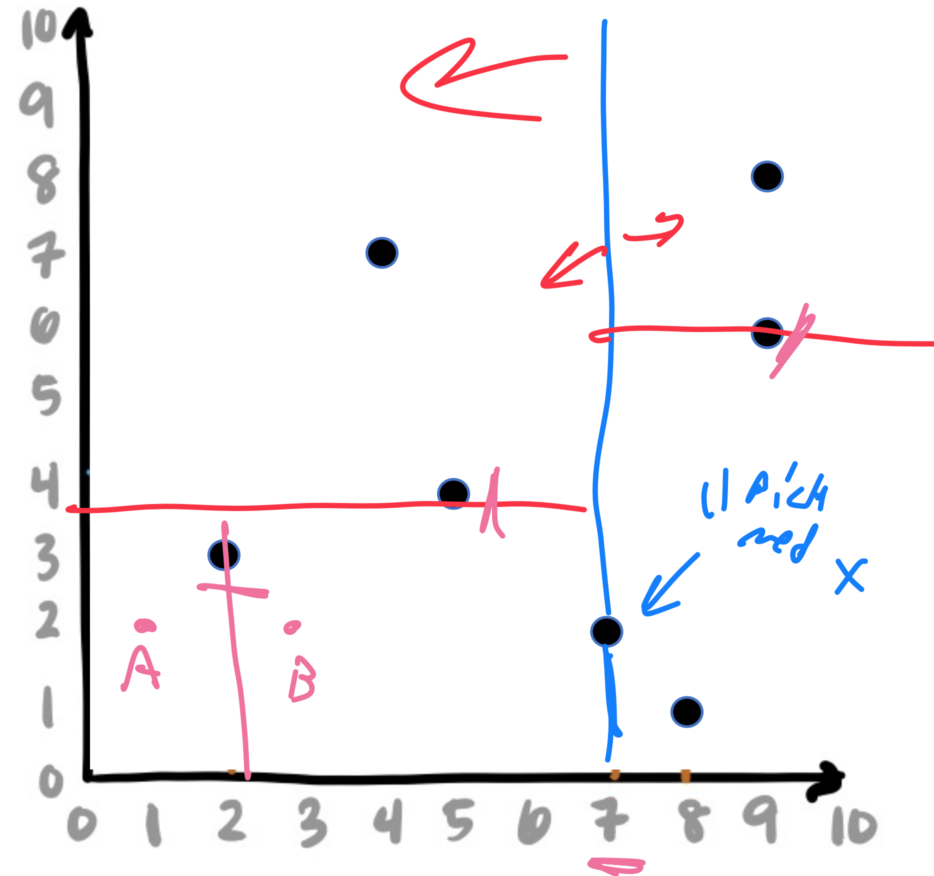
September 27 (AVL analysis)

Nearest Neighbor: k-d tree

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

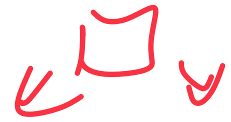
(7,2), (5,4), (9,6), (4,7), (2,3), (8,1), (9,8)





September 29 (KD Tree)

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 _______________ children.

All non-root, internal nodes have _______________ children.

All leaves in the tree are at the same level.

October 6 (BTree Analysis)

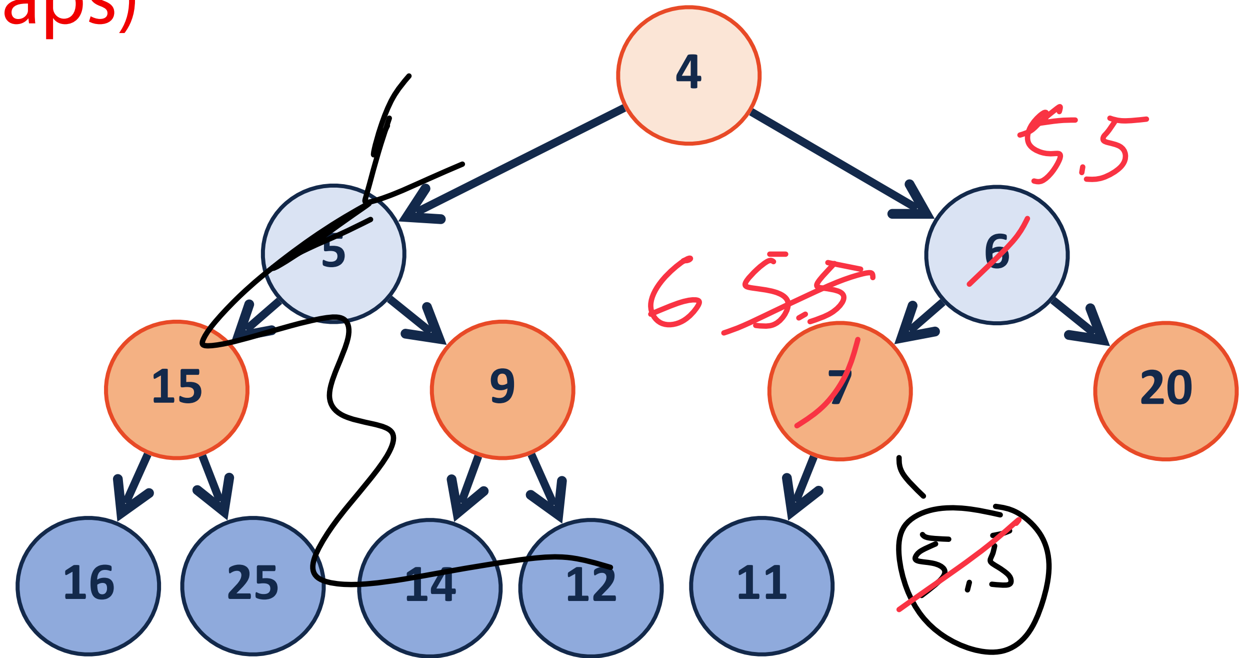
Heaps

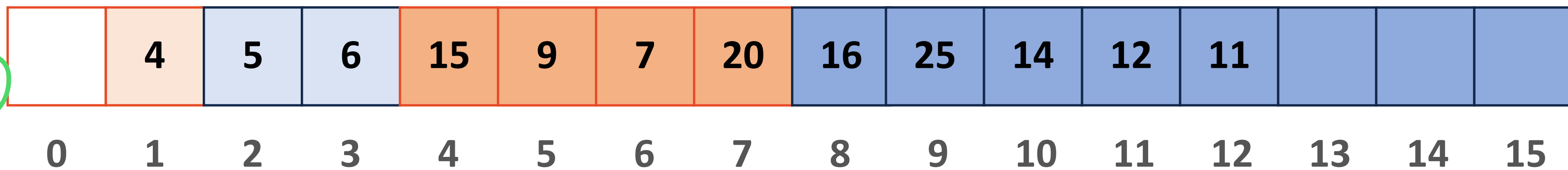
(min)Heap

October 13 (Heaps)

Insert (i)
↳ Array pushBack (i)
↳ Swap until heap again
↳ 'heapify up'

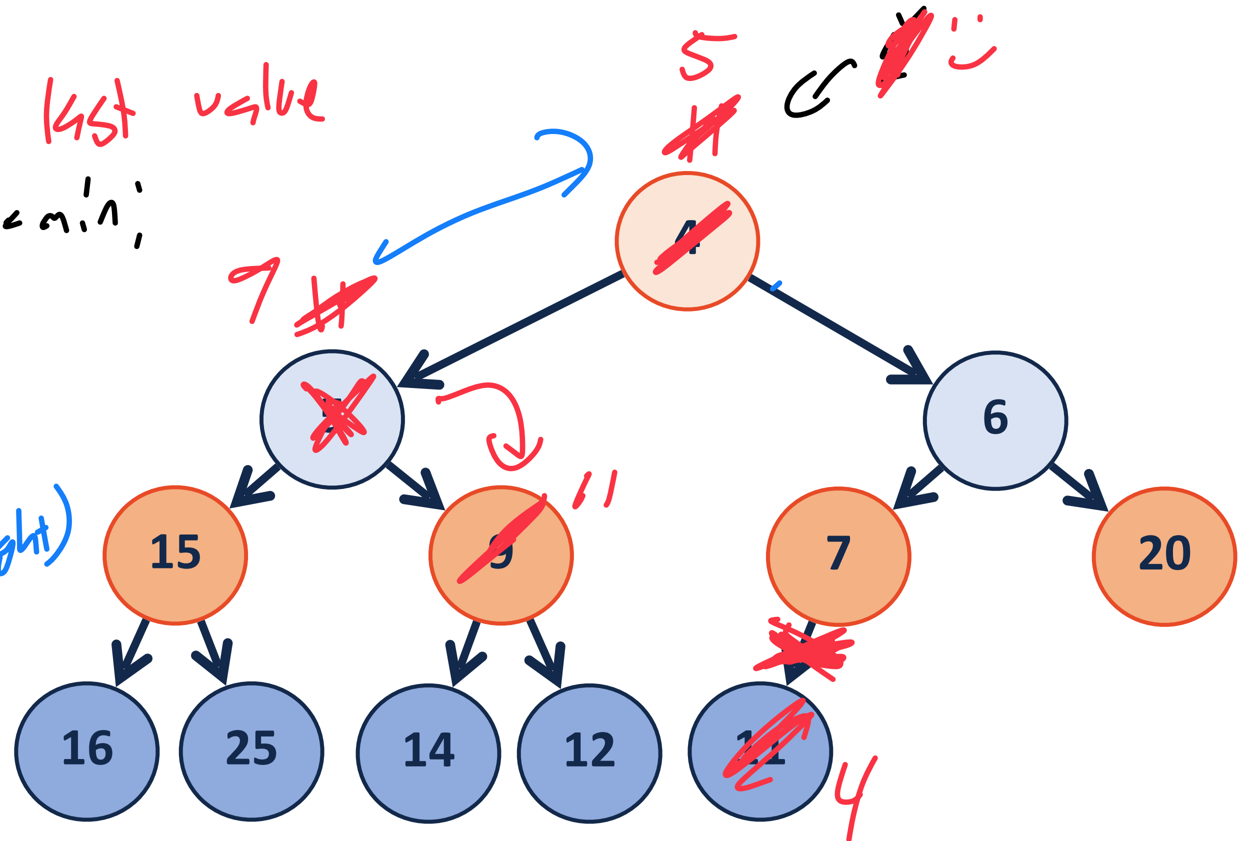
$O(\log n) = O(h)$

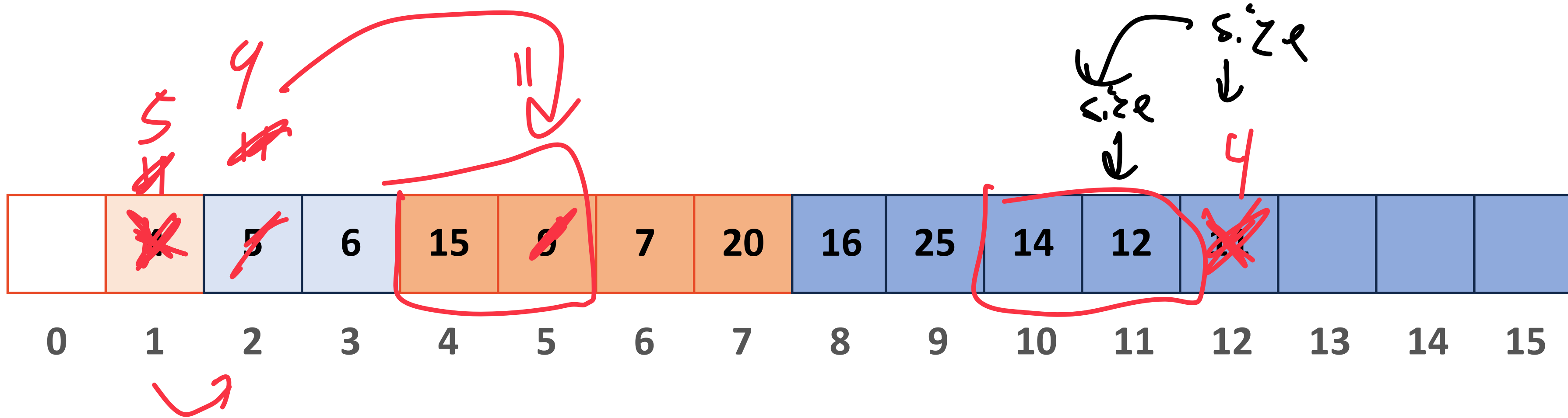




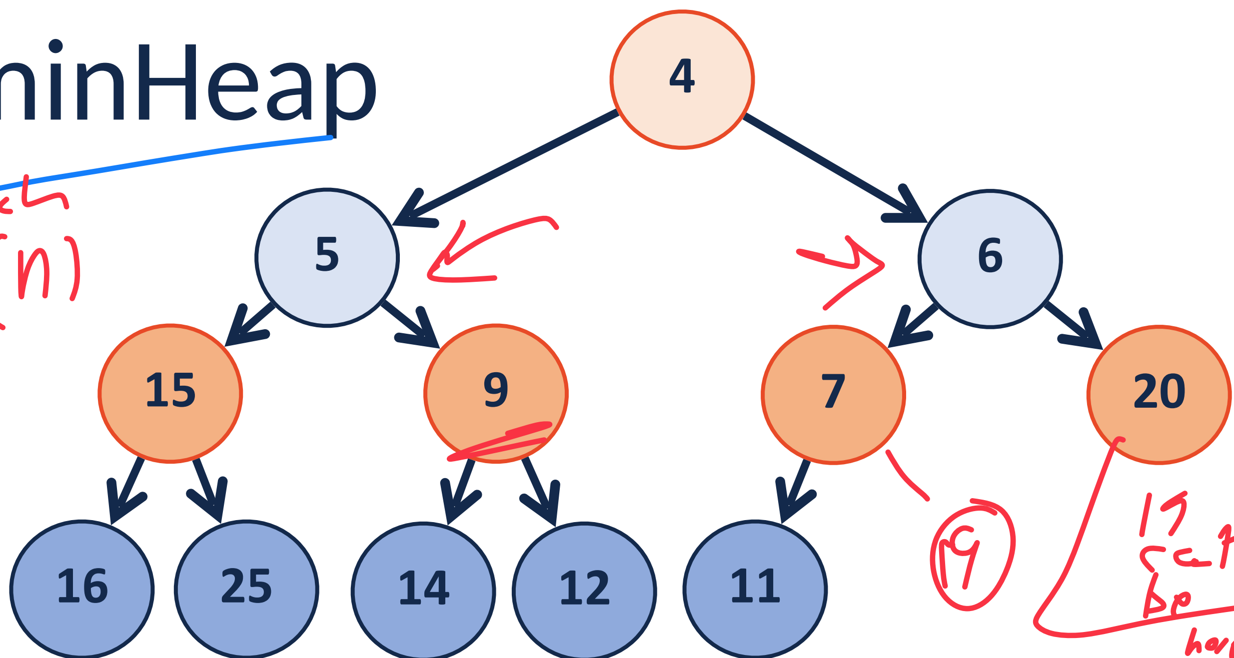
removeMin

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

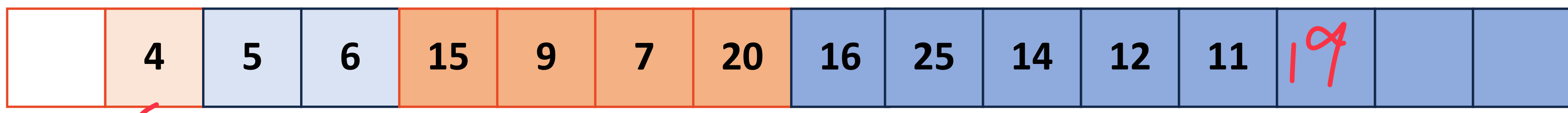




minHeap



1. Construction → O(n)
2. Insert → O(log n)
3. RemoveMin → O(log n)



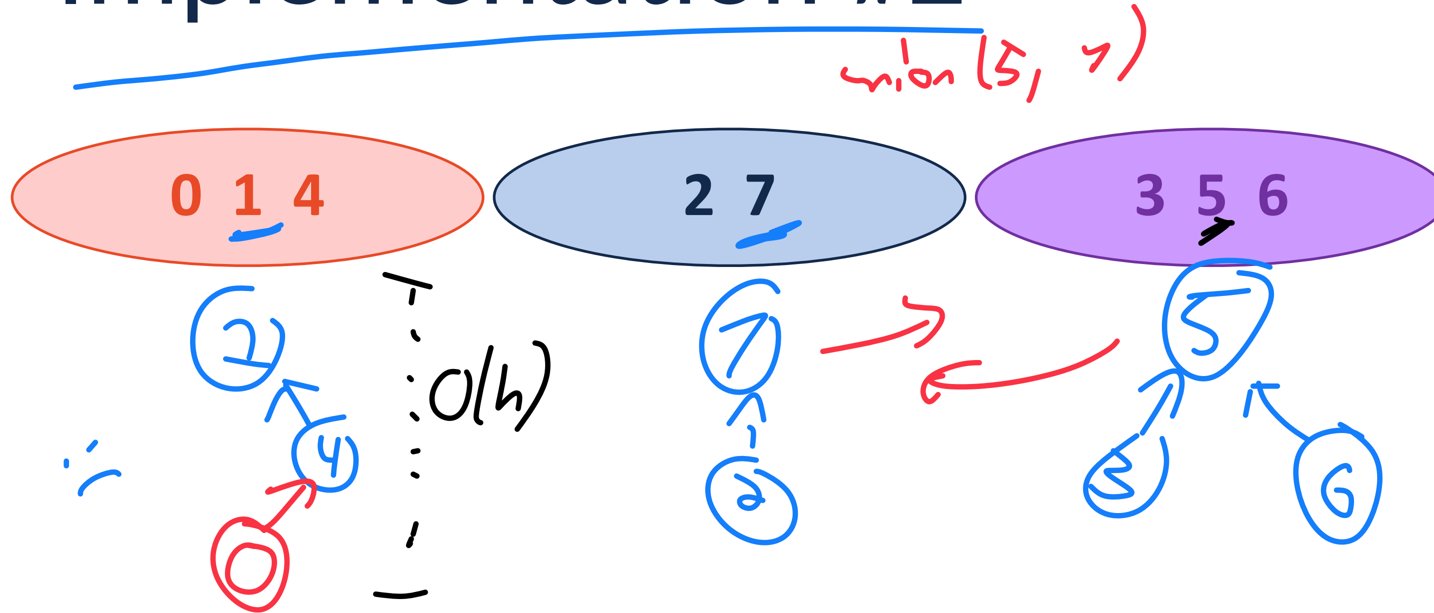
minHeap is a good example of tradeoffs:

October 16 (EC Project / Disjoint Sets)

Disjoint Sets

Implementation #2

October 18 (Sets 2)



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

Find(k): O(h) → A little slower

↳ last class: O(1) w/ array

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

(we choose one value)

↳ last class: O(n)

up tree

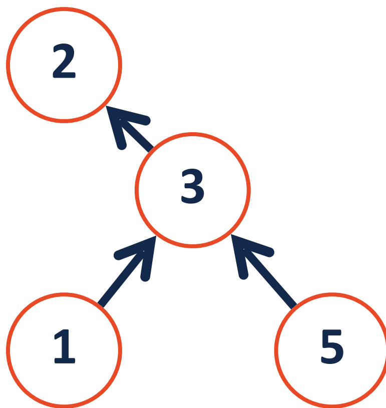
1) All non-canonical elements point to canonical (or 'root')
2) Canonical (root) elements store -1

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)

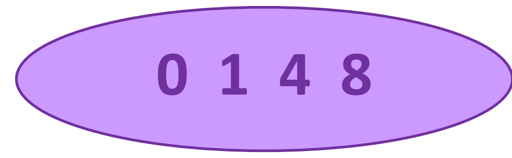
- The value **-1** is our representative element (the root)
- All other set members store the index to a parent of the UpTree

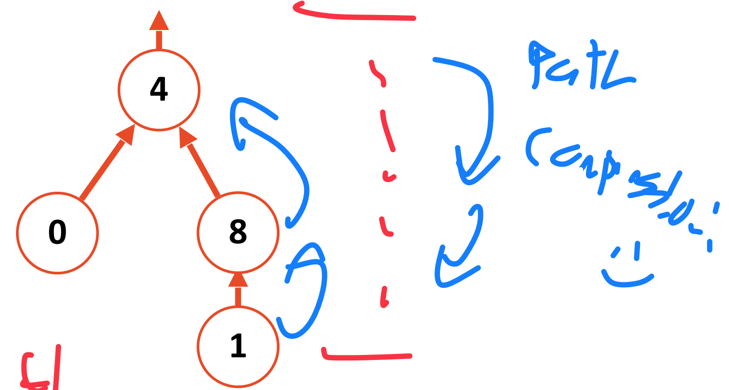


Disjoint Sets Find

Find(1)

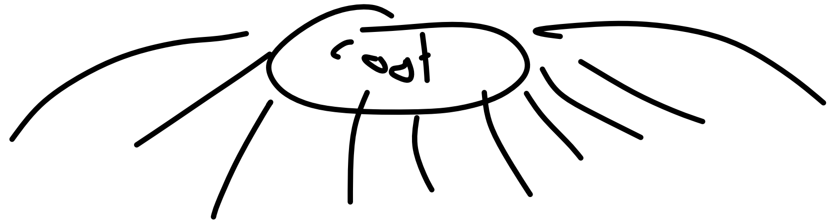
```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else { return find( s[i] ); }
4  }
```





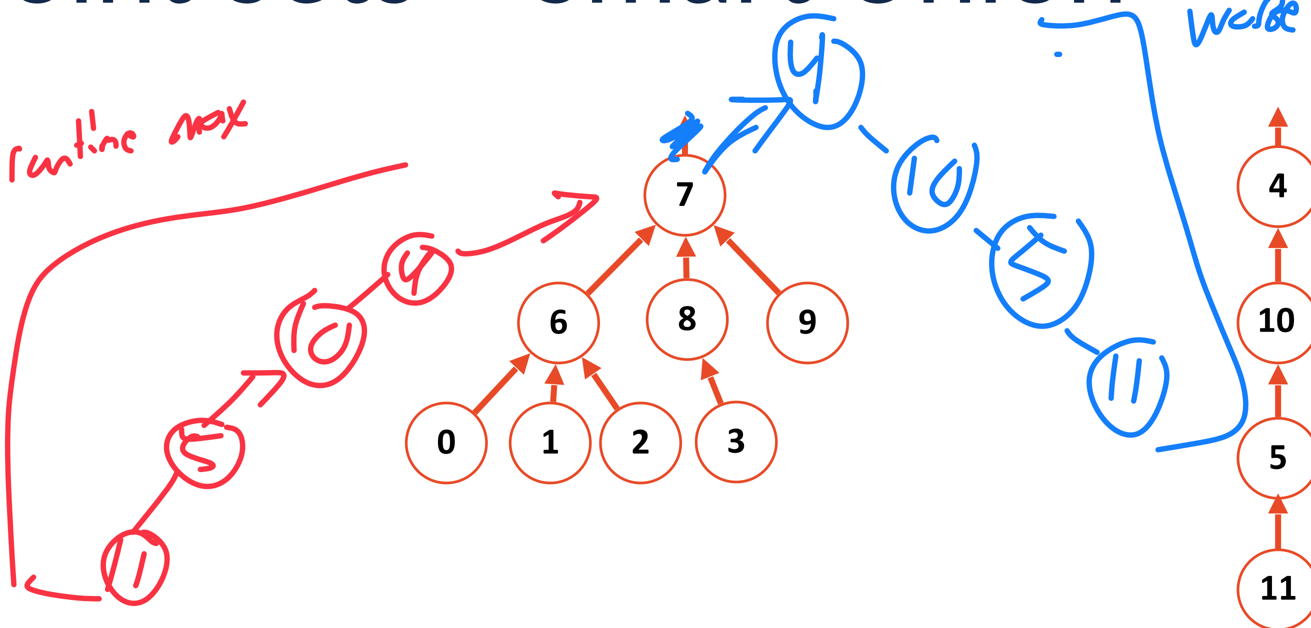
Running time?

What is ideal UpTree?



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: _______________.

Final Result

October 23 (Probability in CS)

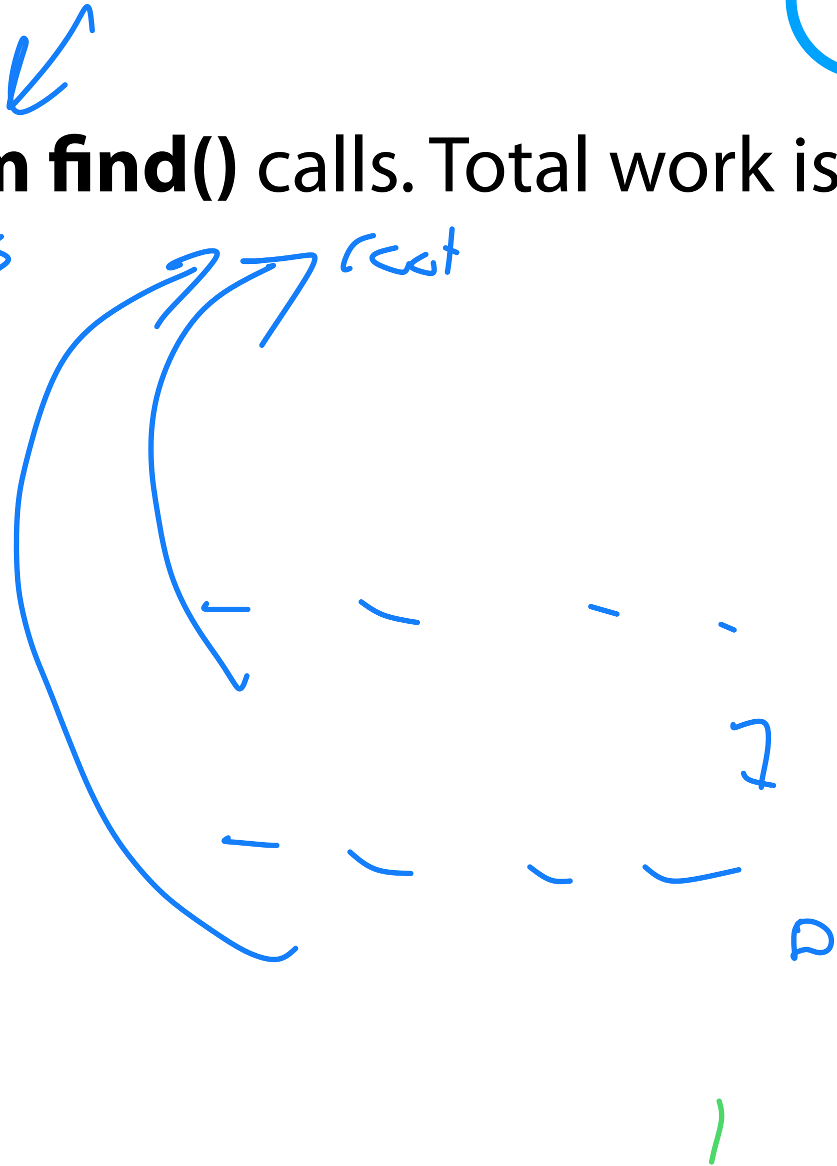
We have **n items** in an Uptree. We make **m find()** calls. Total work is:

cost between buckets + cost inside buckets

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

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

Eventually my finds are just this



Probability in CS

Probabilistic Data Structures

A Hash Table based Dictionary

October 30 (Hashing 2)

User Code (is a map):

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

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

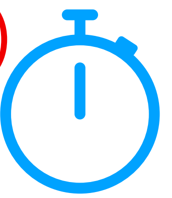
1. A hash function   Data → int

2. A data storage structure   Array

3. A method of addressing *hash collisions*

Hash Table (Separate Chaining)

October 27 (Hashing)



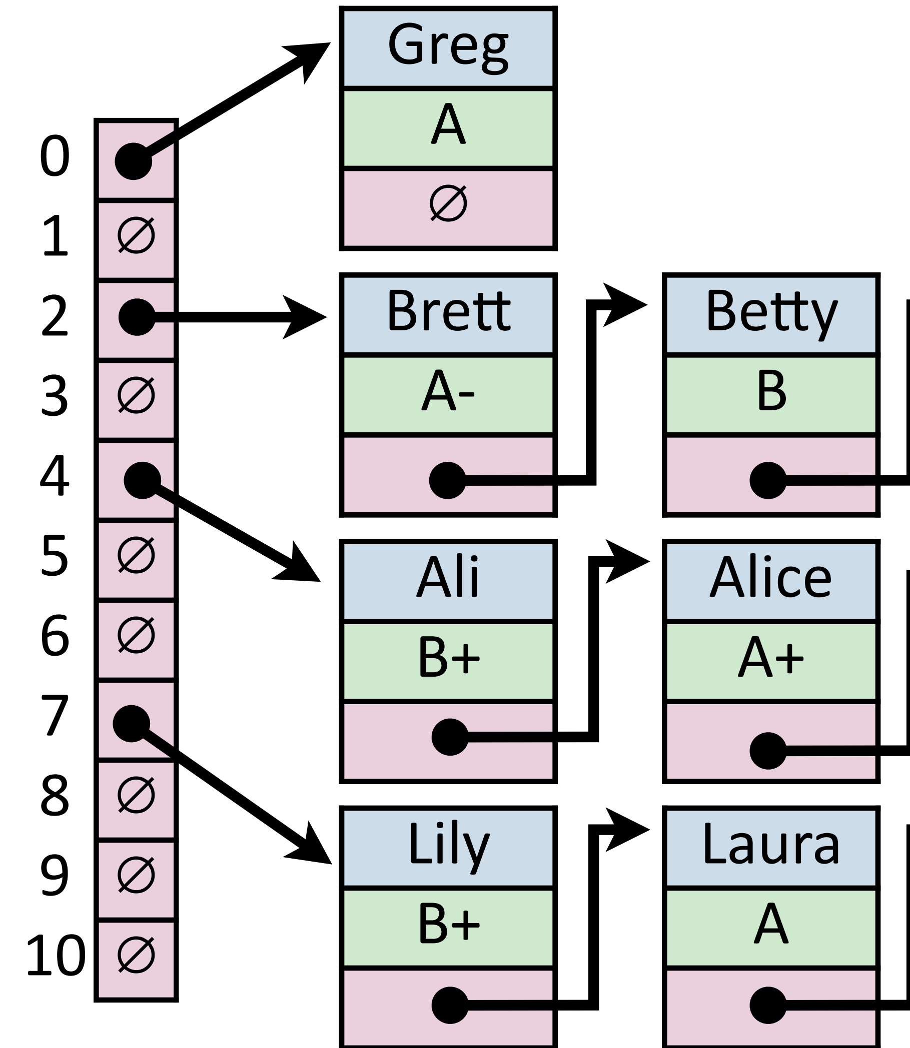
For hash table of size *m* and *n* elements:

Find runs in: O(n) ____________

Insert runs in: O(1) :) ____________

Remove runs in: O(n) ____________

↳ B/c of collisions!



Simple Uniform Hashing Assumption

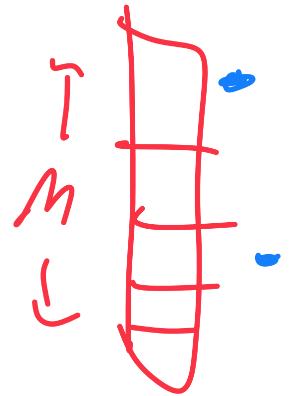
October 30

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

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

Uniform: All keys equally likely to hash anywhere

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



Independent: All keys hash independently of each other

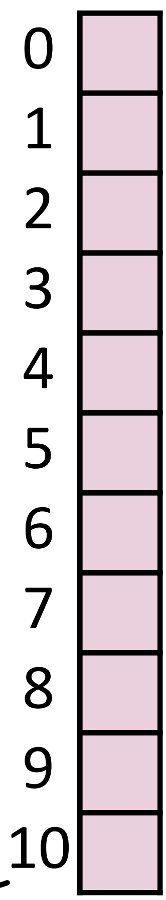
Separate Chaining Under SUHA

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

Find runs in: ____________.

Insert runs in: ____________.

Remove runs in: ____________.



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)**

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

OH: $\alpha \to \infty$

CH: $0 \le \alpha < 1$

Don't let it be 1

Instead, observe:

- As α increases:

Search time → ∞

- If α is constant:

↳ Runtime is constant

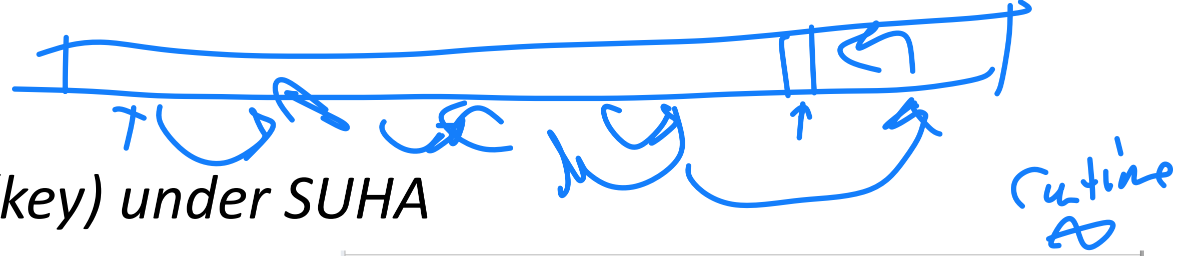
If n = 100, m > 100

m = 200

November 1 (Bloom Filter)

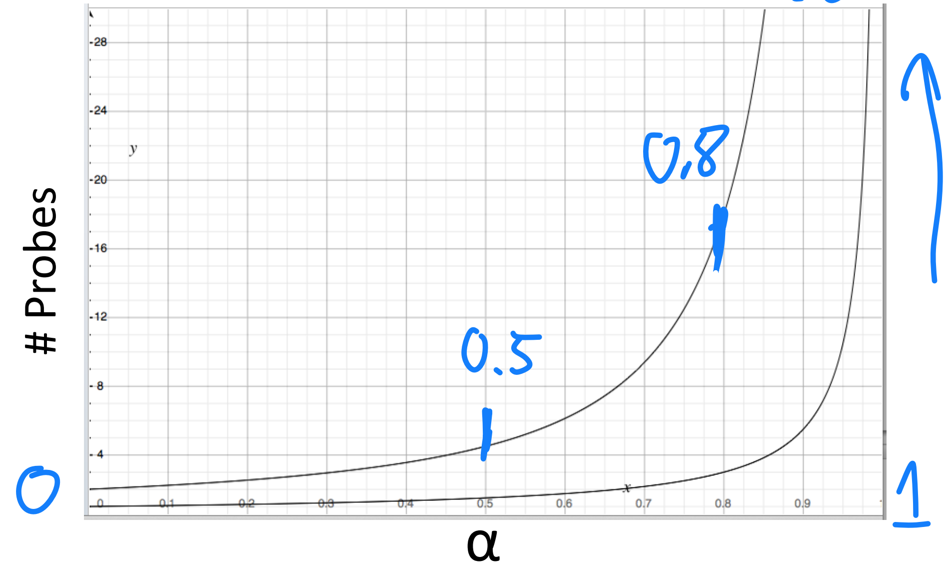
Running Times

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



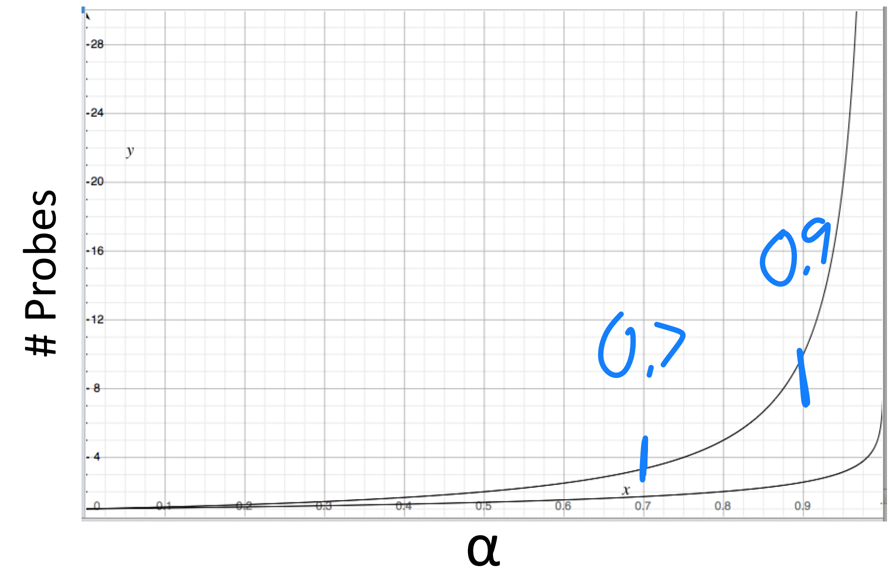
Linear Probing:

- 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

log 2

	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

November 3 (BF 2)

Bloom Filter

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

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

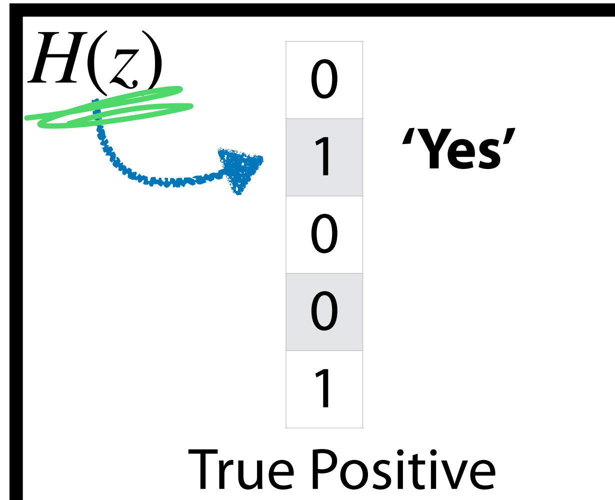
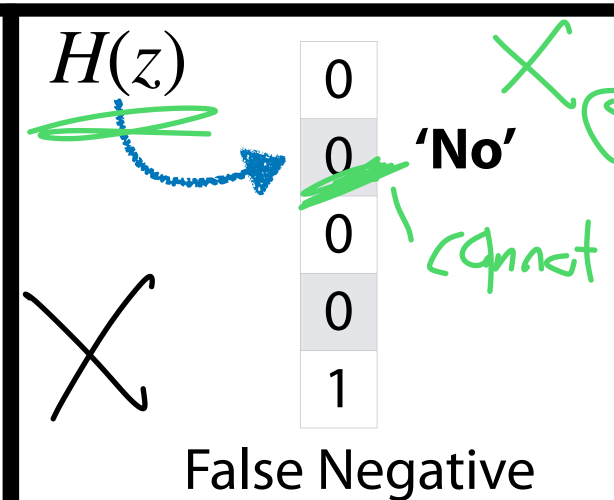
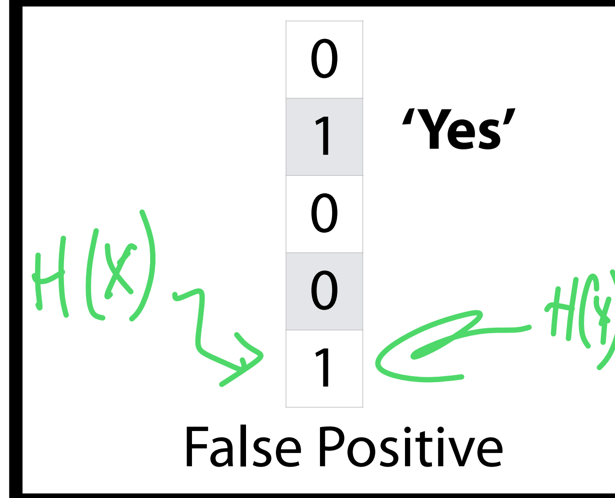
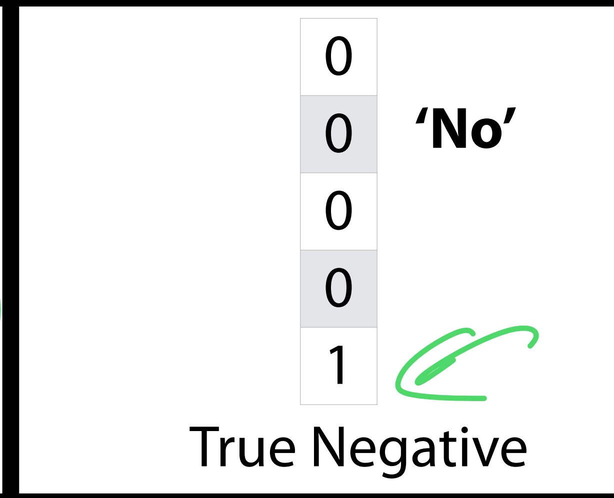
Delete is not possible (yet)!

In real world possible

↳ One sided error b/c we don't delete

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

predicted

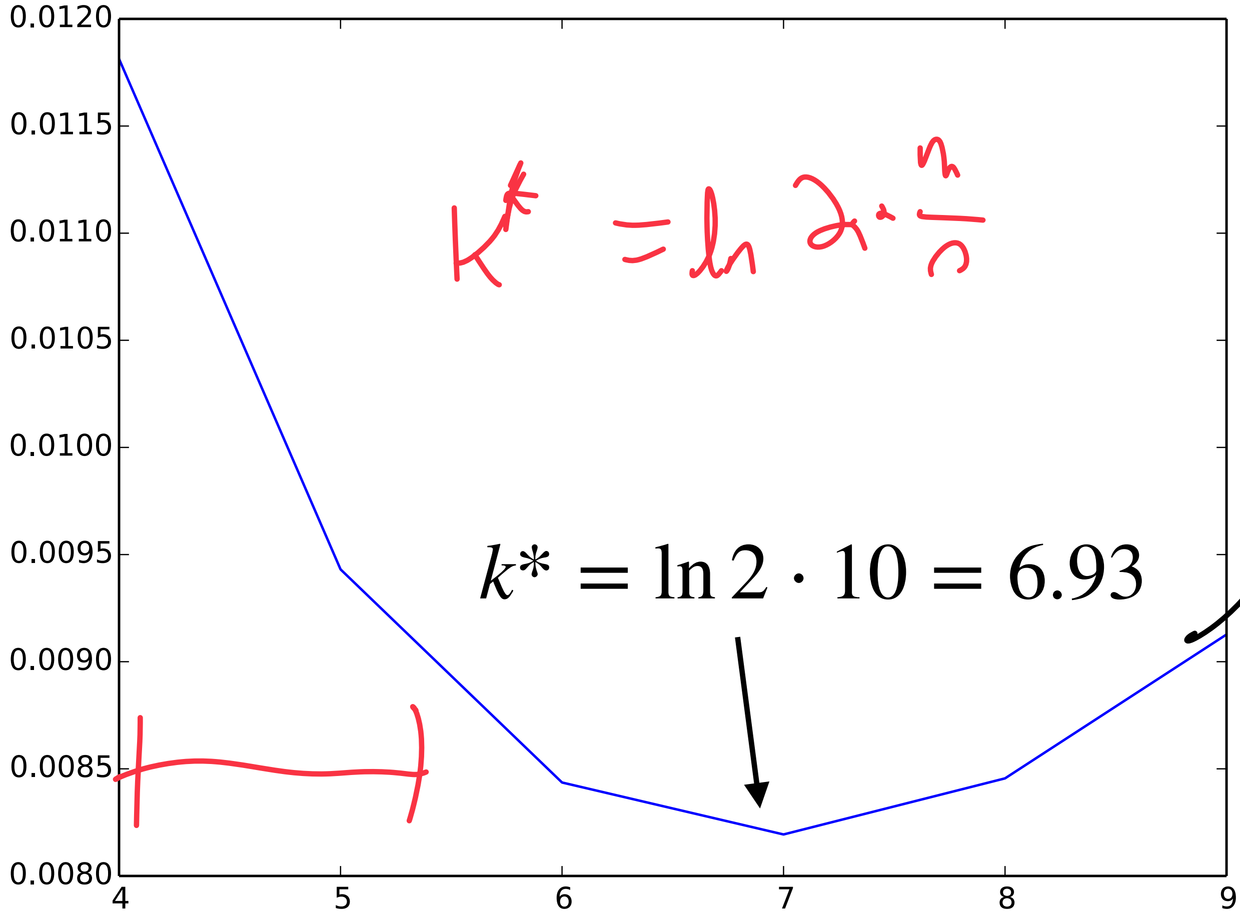
	Bit Value = 1 (Yes)	**Bit Value = 0** (No)
Item Inserted		
Item NOT inserted		

Actual truth

One sided error

Bloom Filter: Error Rate

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



k

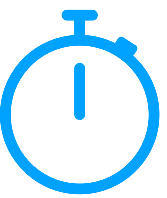
k is too small
not enough random trials

Saturate my filter

Figure by Ben Langmead

Cardinality Estimation

November 6 (Cardinality)



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:

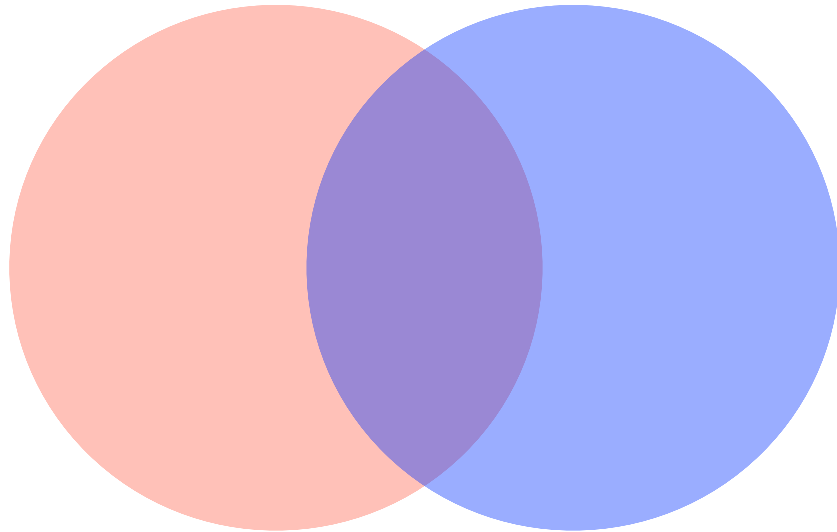
$$95 \approx 1000/(N + 1)$$

$$N + 1 \approx 10.5$$

$$N \approx 9.5$$

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!

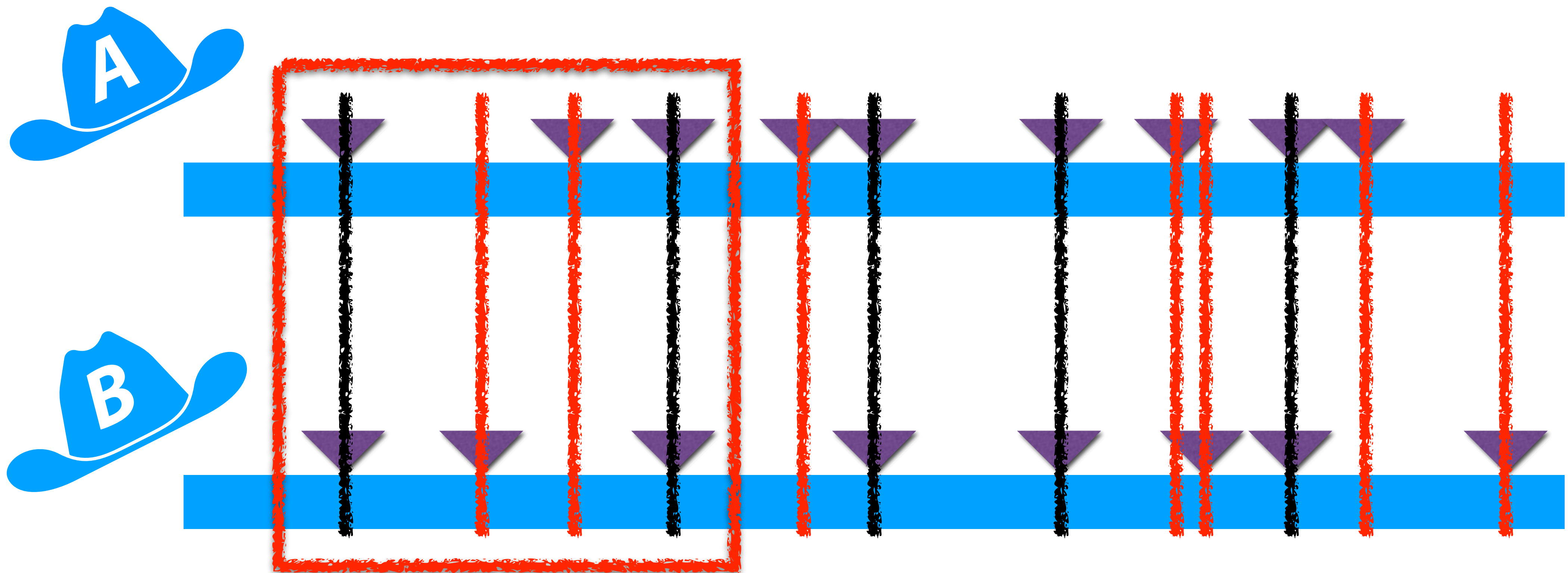
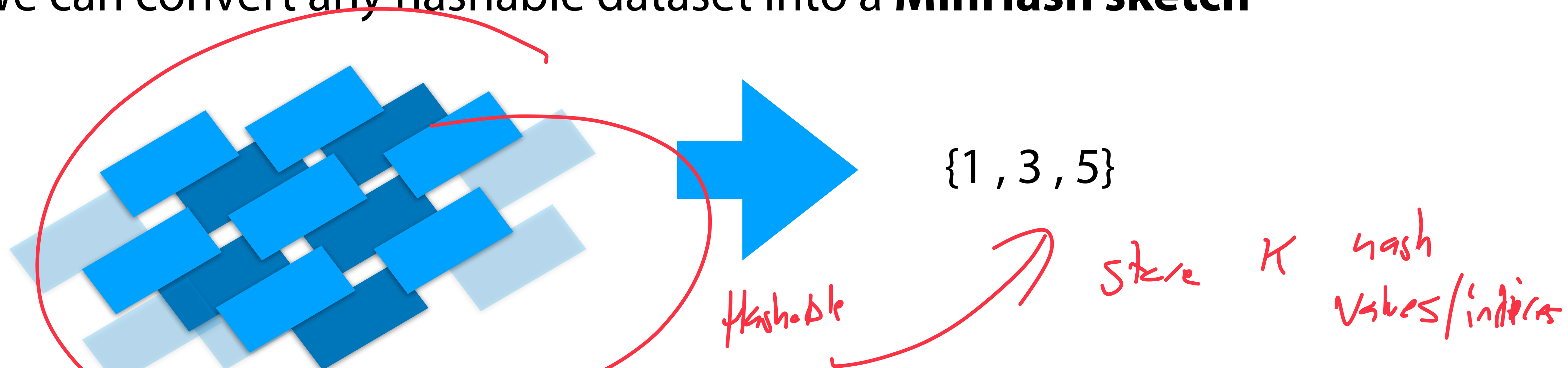


Image inspired by: Ondov B, Starrett G, Sappington A, Kostic A, Koren S, Buck CB, Phillippy AM. **Mash Screen: high-throughput sequence containment estimation for genome discovery**. *Genome Biol* 20, 232 (2019)

November 10 (Graphs)

MinHash Sketch

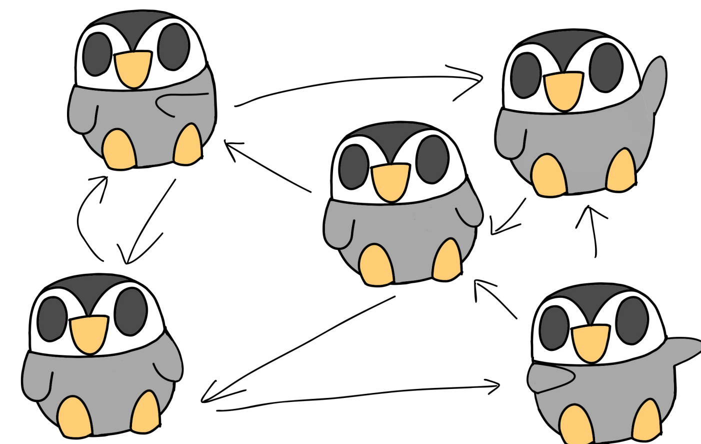
We can convert any hashable dataset into a **MinHash sketch**



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)

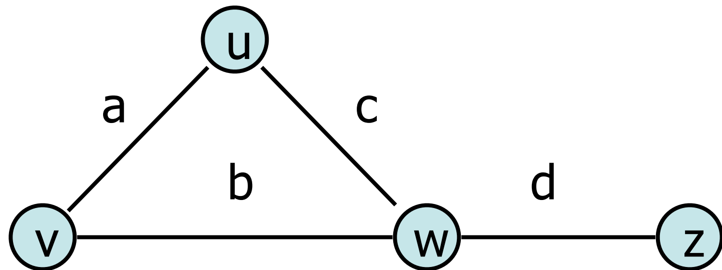
Graphs

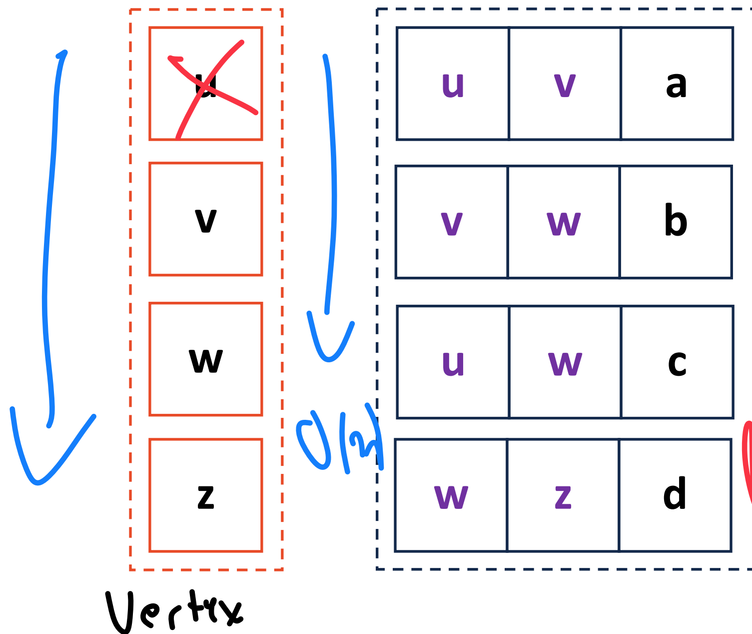


November 15 (Graph Implementation 2)

Graph Implementation: Edge List

|V|= n, |E|= m





insertVertex(K key):

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

removeVertex(Vertex v):

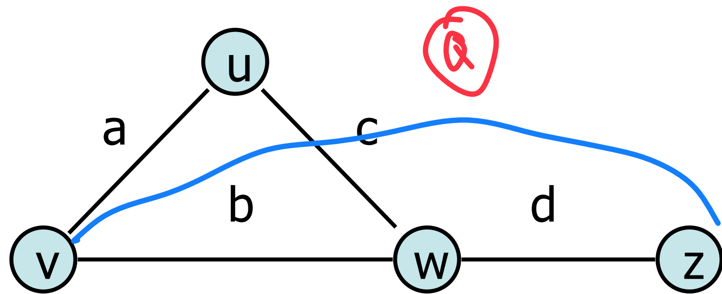
removeEdge(Vertex v1, Vertex v2, K key):

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m



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

Vertex list: u, v, w, z

insertVertex(K key):

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

removeVertex(Vertex v):

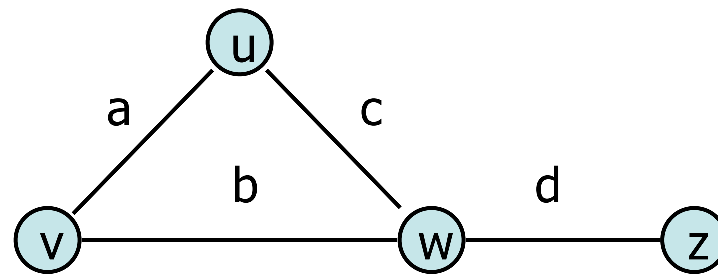
removeEdge(Vertex v1, Vertex v2, K key):

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

Adjacency List

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



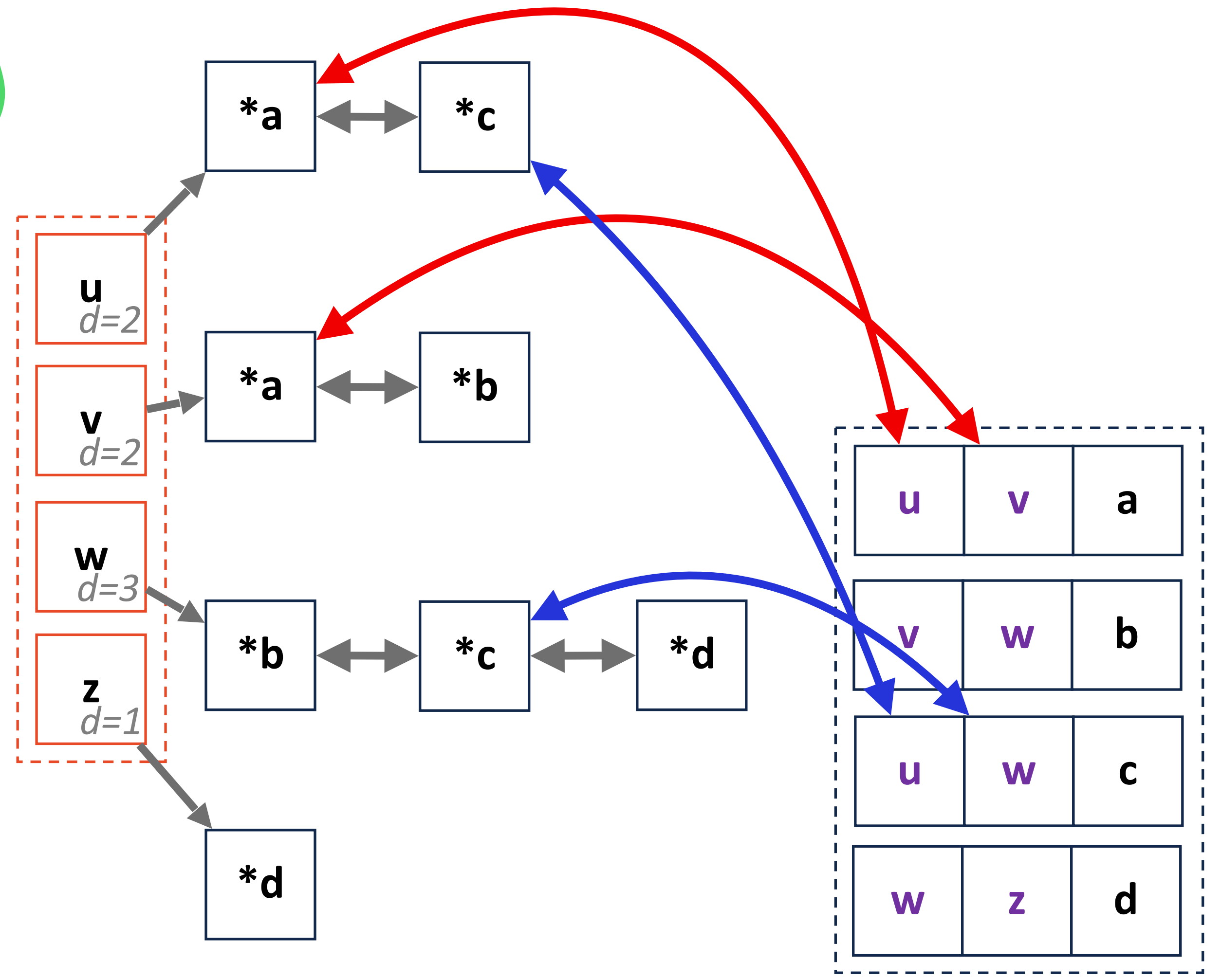
Adj List Node: Bidirectional LL (w/ tail pointer)

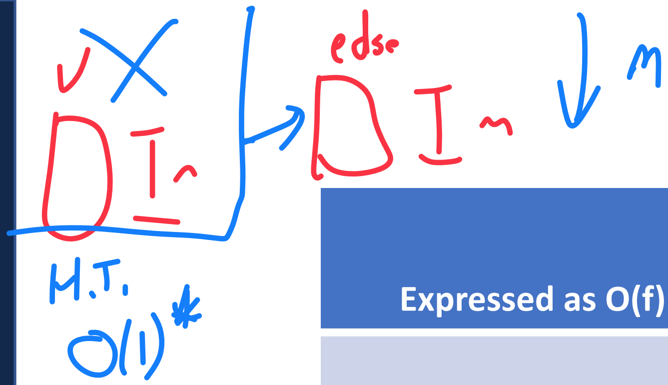
3 pointers

Prev	Edge	Next

Edge List: Edge list + 2 pointers

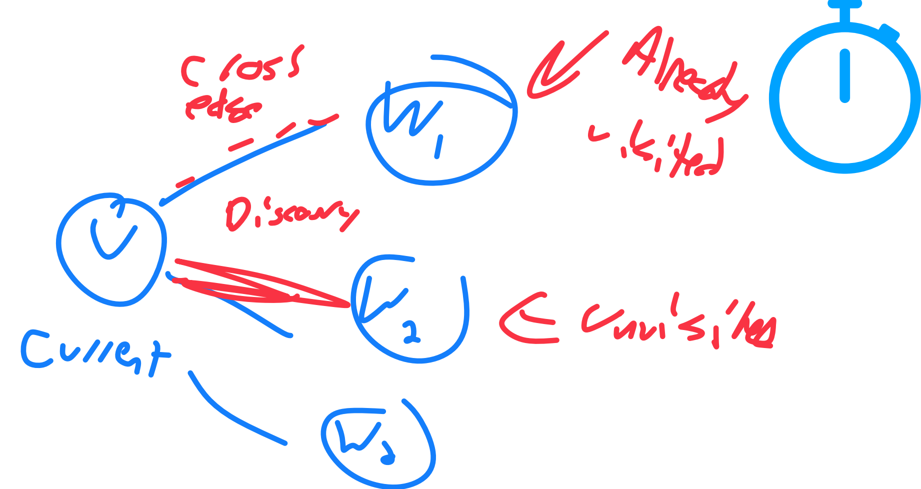
V1	V2	Weight
*V1	*V2	





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

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))

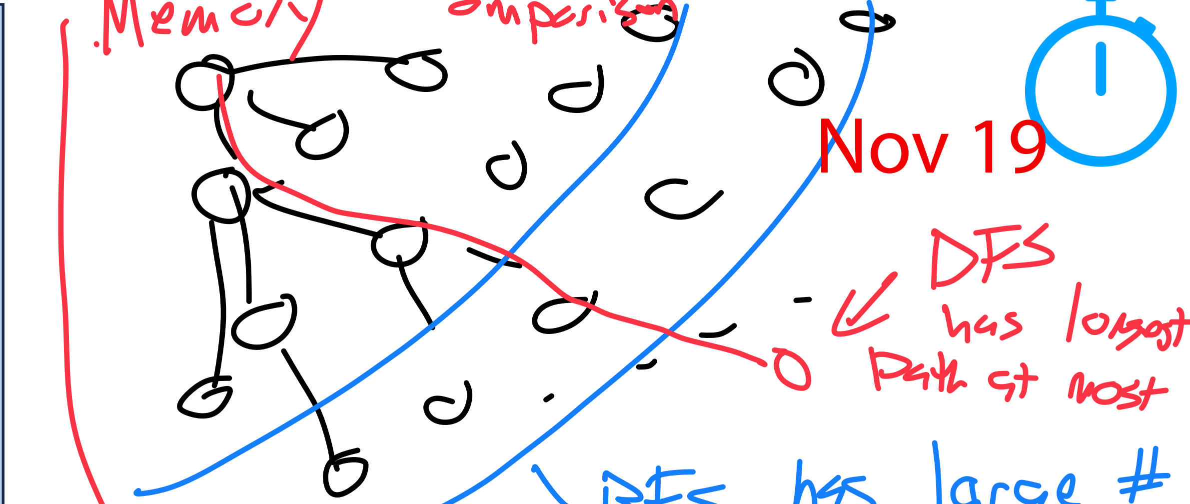


```
1  BFS(G):
2    foreach (Vertex v : G.vertices()):
3      setPred(v, NULL)
4      setDist(v, -1)
5
6    foreach (Edge e : G.edges()):
7      setLabel(e, UNEXPLORED)
8
9    foreach (Vertex v : G.vertices()):
10     if getDist(v) == -1:
11       BFS(G, v)
```

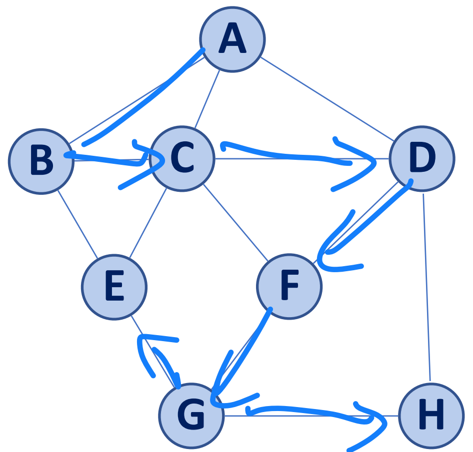
```
12 BFS(G, v):
13   Queue q
14   setDist(v, 0)
15   q.enqueue(v)
16
17   while !q.empty():
18     v = q.dequeue()
19
20     foreach (Vertex w : G.adjacent(v)):
21       if( getDist(w) == -1):
22         setLabel((v, w), DISCOVERY)
23         setPred(w, v)
24         setDist(w, v + 1)
25         q.enqueue(w)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

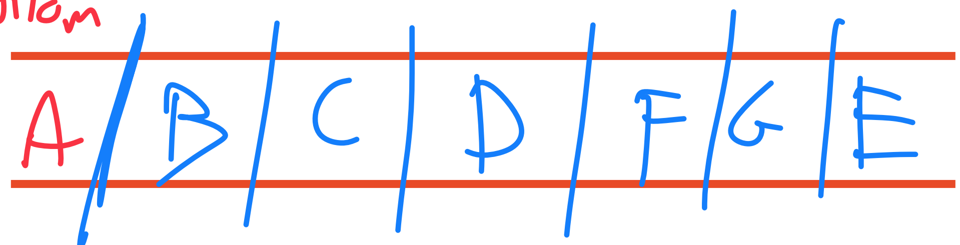
November 17
(Graph Traversal)

```
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)
```

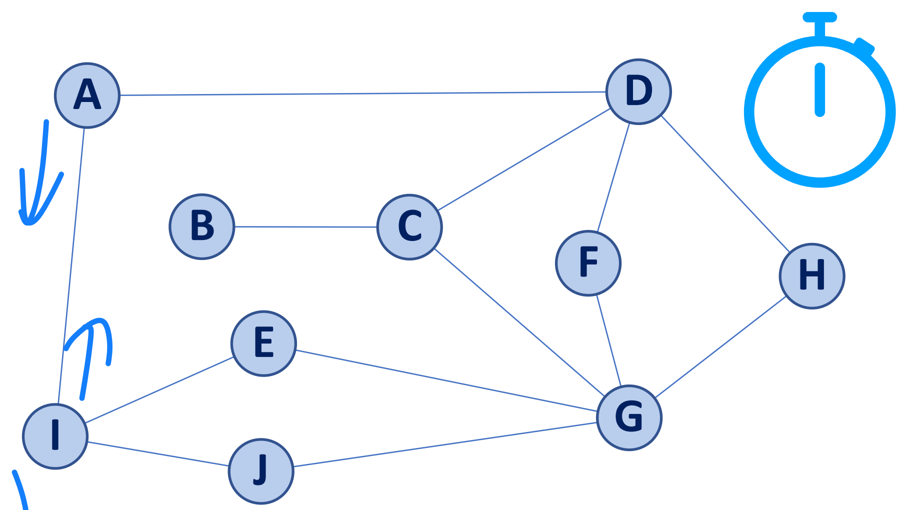




A → B, C, D
stop!
DFS(B) → DFS(C) → DFS(D)

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: $O(m)$

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

$O(m)$

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

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

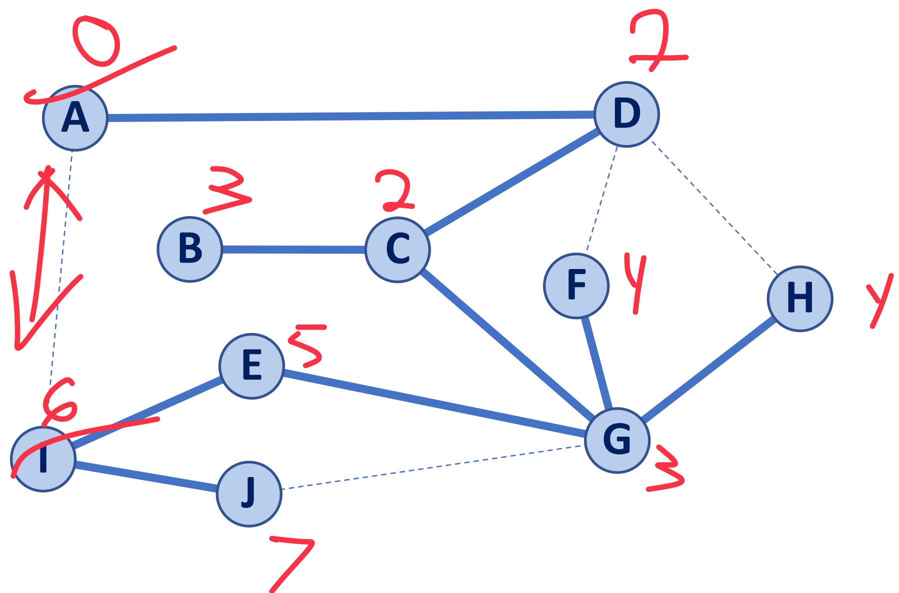
BFS Observations

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



Discovery Edge

Back Edge

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

Do we still make a spanning tree?

↳ Yes it makes a MST*

* on unweighted graphs

Does distance have meaning here?

↳ Has no real meaning!

Do our edge labels have meaning here?

↳ Back edge still implies cycle

Kruskal's Algorithm

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

```
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)

Construction

get/remove Min

1) Disjoint Set & Priority Queue

* Destructive

Sorted array persist

Prim's Algorithm

Sparse Graph: O(n) ≤ O(m) ≤ O(n²)

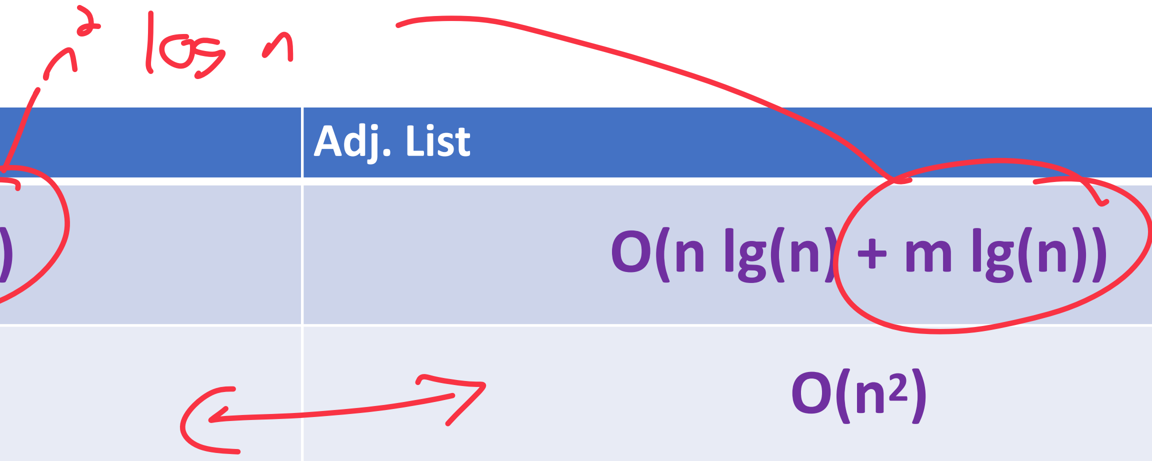
↳ m ~ n → O(n log n)

↳ adj list heap is best

Dense Graph: m ~ n²

↳ 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	$O(n^2)$	$O(n^2)$

Dijkstra's Algorithm (SSSP)

What is the running time of Dijkstra's Algorithm?

```
    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
```

Dijkstra's Algorithm (SSSP)

Dijkstras Algorithm works only on non-negative weights* efficiently

No neg cycles allowed

Optimal implementation:

Fib heap

on dense graph — unsorted list — ties

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?

```
   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]
```