Data Structures
Review

CS 225
Brad Solomon

December 9, 2024

!!

You got this!

Department of Computer Science

Announcements

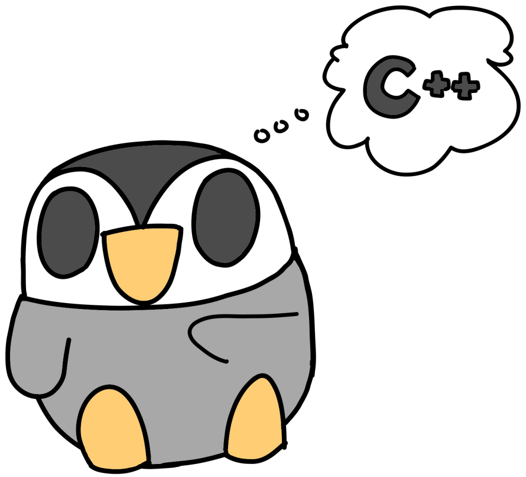
Fill out ICES forms!

Interested in being a CA? Apply for CS 225 or CS 277!

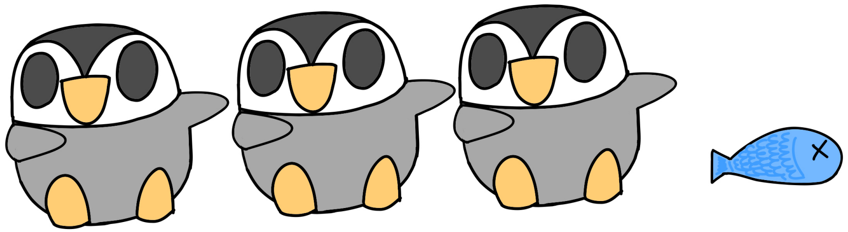
https://opportunities.cs.illinois.edu/courses/positions/

Material covered here is not only material in class!

Represents only an attempt to provide some helpful resources.


C++

Lists



List Implementation

September 9 (Array List Lecture)

	Singly Linked List	Array
Look up **arbitrary** location ↳ index	O(n)	O(1)
Insert after **given** element ↳ ref pointer	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	Find O(n) change O(1) O(n)	Find O(1) change O(n) O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

Special cases

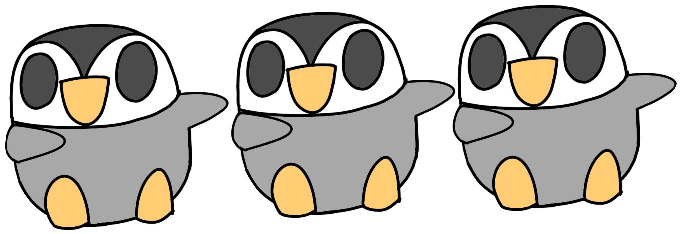
insert / remove Front O(1)

if array not full always amortized

insert Back remove is O(1)*

Lists

November 6 (Review Lecture)



The not-so-secret underlying implementation for many things

	Singly Linked List	Array
Look up **arbitrary** location	O(n)	O(1)
Insert after **given** element	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)

Special Cases:

insert Front O(1)

Insert Back O(1)*

Stack and Queue

November 6 (Review Lecture)

Taking advantage of special cases in lists / arrays

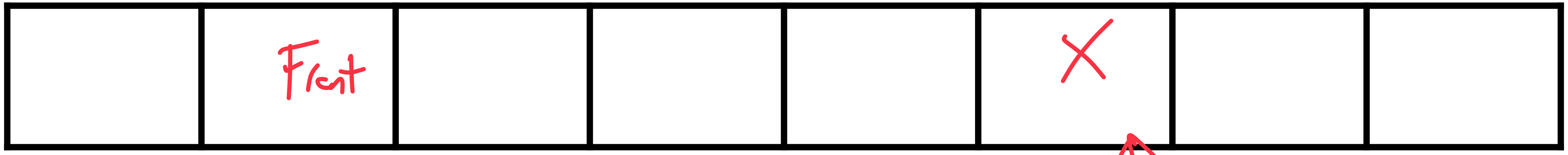
Queue

O(1) access

O(n) change value

insert / remove Back

O(1)*



O(1)*

Don't move items!

Back

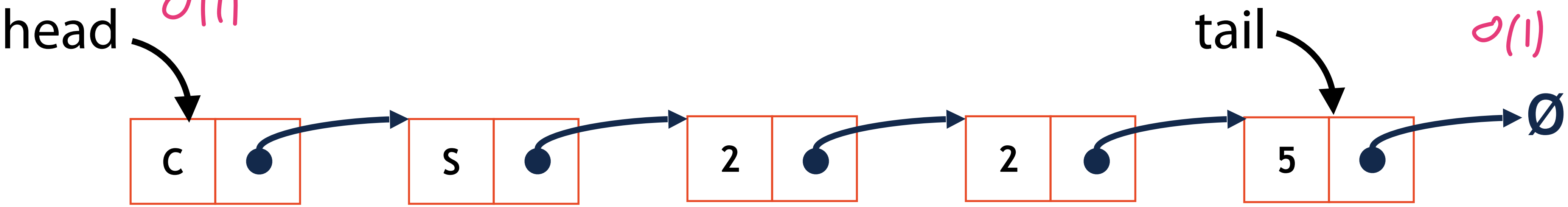
O(1)*

Front

O(1)

Back

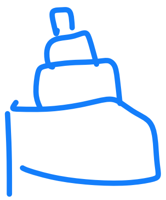
O(1)



Stack ADT

September 11 (Quacks Lecture)

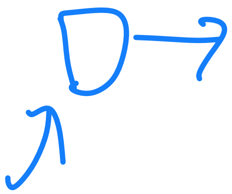
- [Order]: Last in first out (LIFO)



- [Implementation]: Trivially as vector or LL

student Q

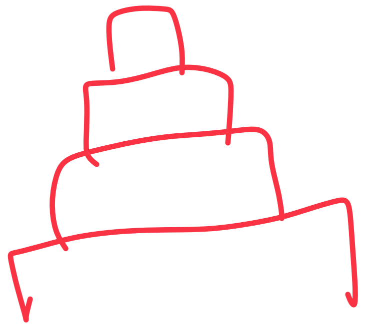
Front is top



- [Runtime]: $\boxed{O(1)^*}$

* if array is not full
if array is full, amortized still says $O(1)$

Stack ADT



- [Order]: LIFO
- [Implementation]: Array (such as std::vector)
- [Runtime]: O(1)* Push and Pop

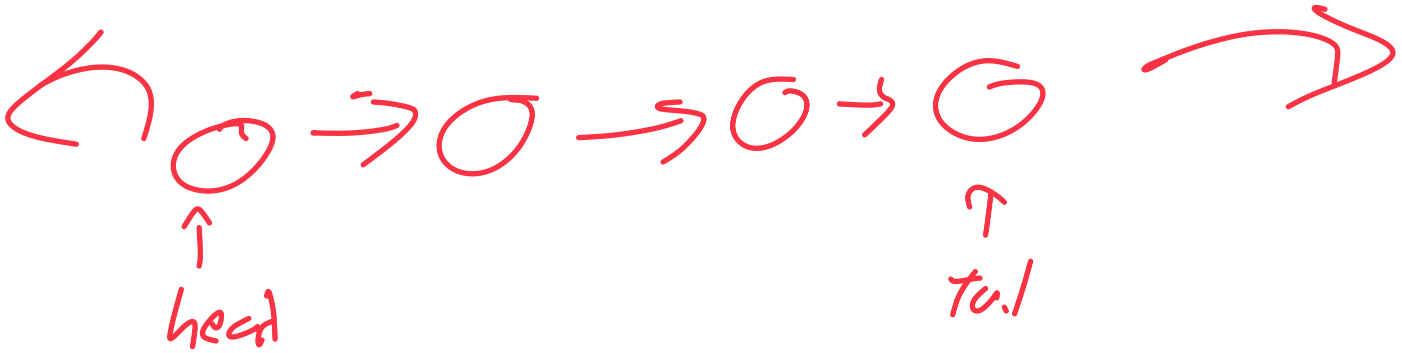
September 13 (Trees Lecture)

Queue ADT

- [Order]: First in First out (FIFO)
- [Implementation]: Vector / dequeue → LL is possible easily
- [Runtime]: O(1) *

September 11 (Quacks Lecture)

Queue ADT



- [Order]: FIFO

- [Implementation]: Circular Queue as Array

- [Runtime]: O(1)

September 13 (Trees 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 ++()` – move to next item

`const T & operator *()` ← return the data/value at current pos

`bool operator !=(const Iterator &)` ← check if iterators are equal

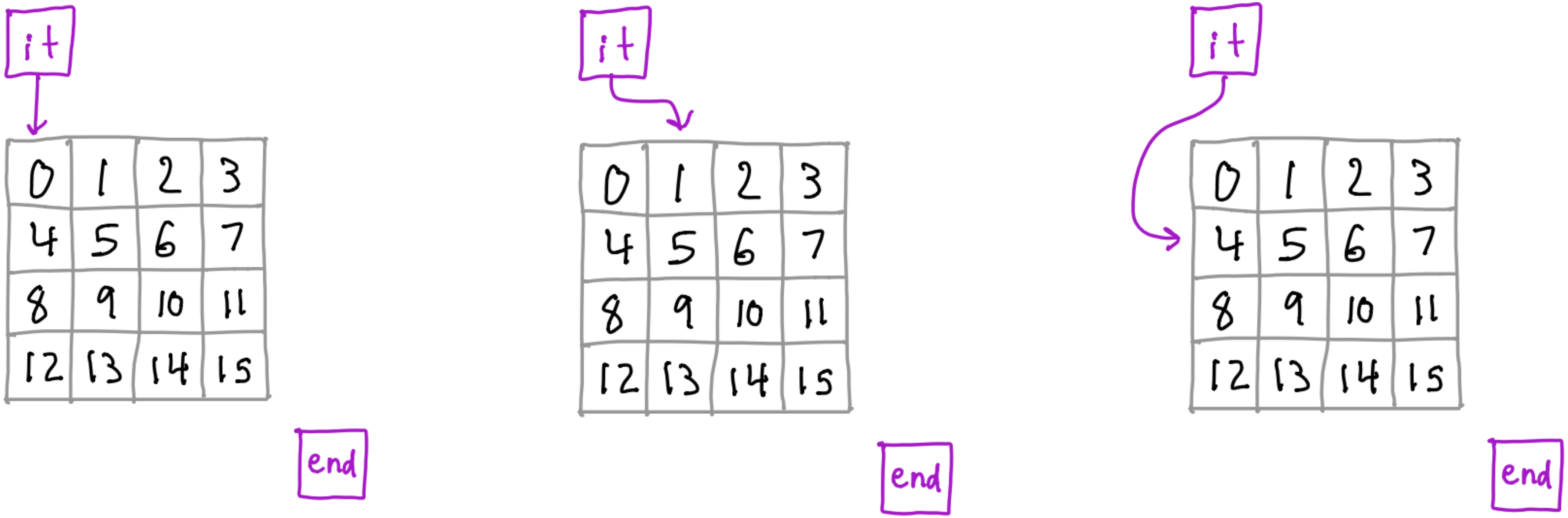
Iterators

Here is a (truncated) example of an iterator:

```
template <class T>
class List {

    class ListIterator : public std::iterator<std::bidirectional_iterator_tag, T> {
      public:

        ListIterator& operator++();

        ListIterator& operator--()

        bool operator!=(const ListIterator& rhs);

        const T& operator*();
    };

    ListIterator begin() const;

    ListIterator end() const;
};
```

```
std::vector<Animal> zoo;


/* Full text snippet */

  for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }


/* Auto Snippet */

  for ( auto it = zoo.begin(); it != zoo.end; ++it ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }

/* For Each Snippet */

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

Iterators (225 Webpage Resources)

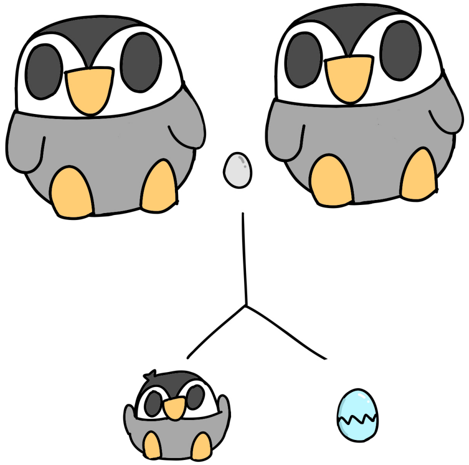


https://courses.grainger.illinois.edu/cs225/fa2024/resources/iterators/

Trees

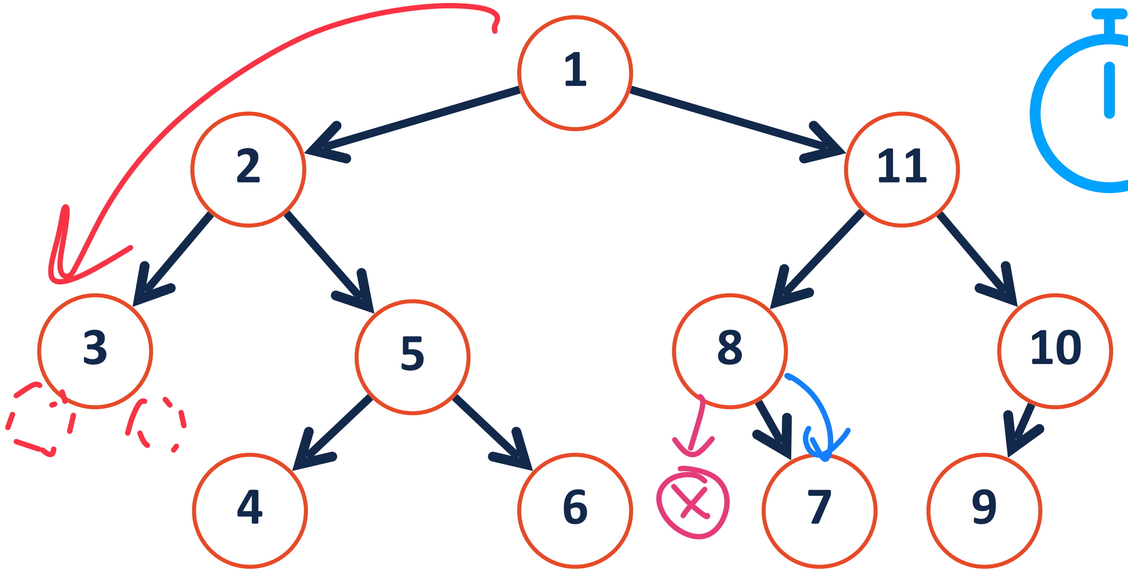
↳ Binary tree!

↳ complete / full / perfect



Tree Traversals

September 18 (Tree Traversal)



root

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

root

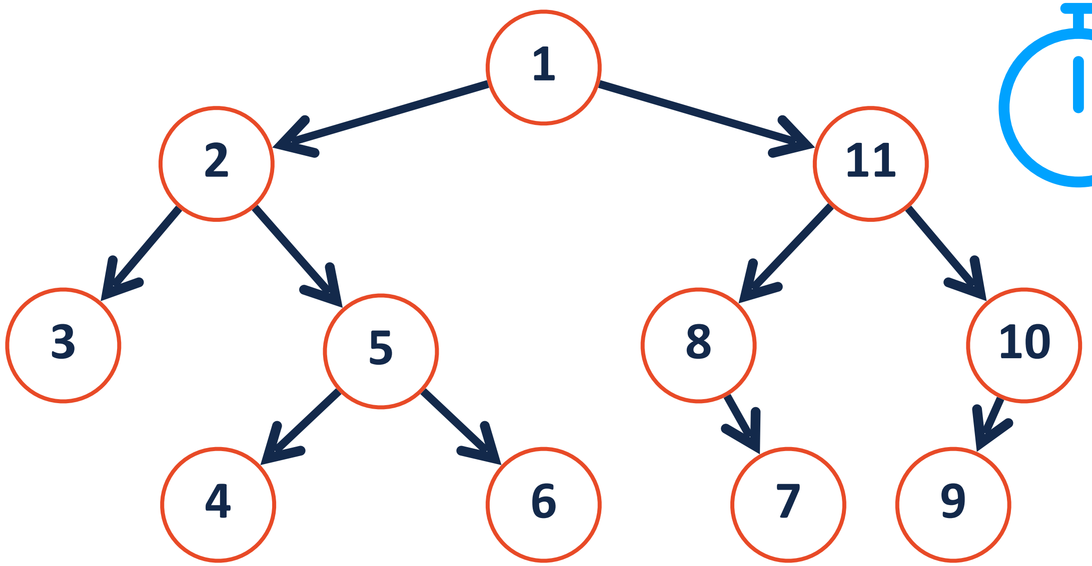
In-order: 3 2 4 5 6 1 8 7 11 9 10

Left than right
Left most child

root

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

Depth First Search

September 20 (BST Lecture)

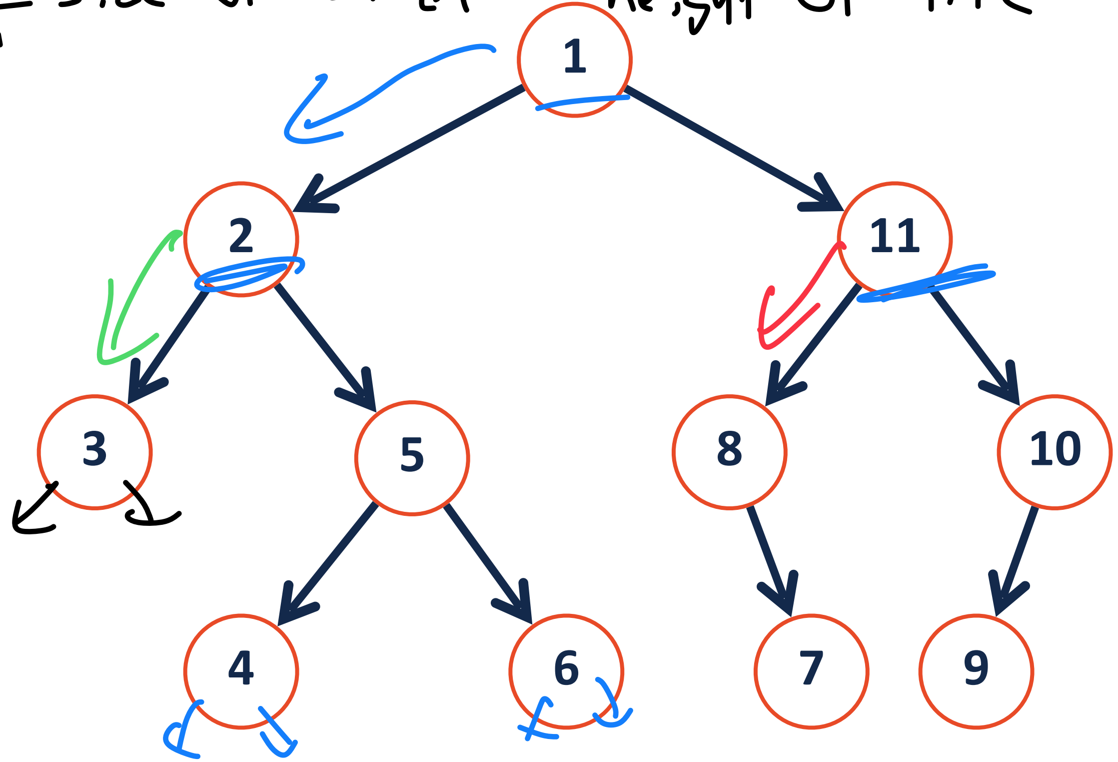
Explore as far along one path as possible before backtracking

Make a stack, initialize to root

While stack not empty

- Pop the top element (as tmp)
- Print tmp
- Push tmp → right
- Push tmp → left

Size of stack ≈ height of tree



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

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

Depth First Search

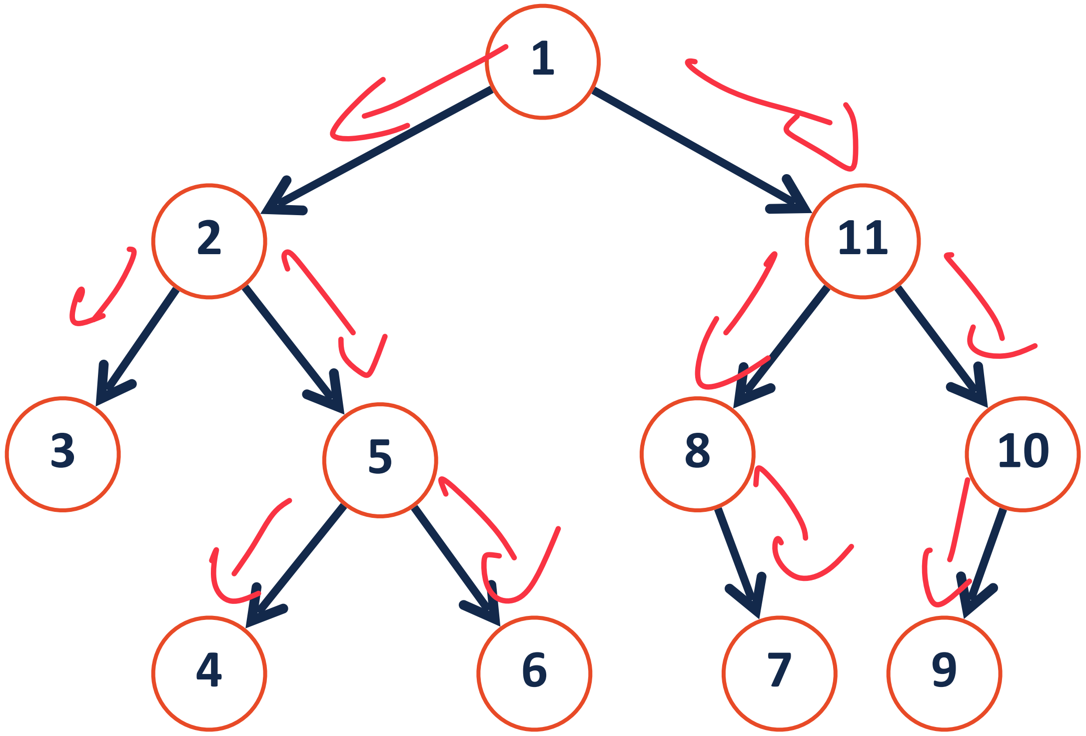
Explore as far along one path as possible before backtracking

Make a stack initialized with root

While stack isn't empty:

- Pop top element (as `tmp`)
- Print `tmp`
- Push `tmp->right` to stack
- Push `tmp->left` to stack

LIFO



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

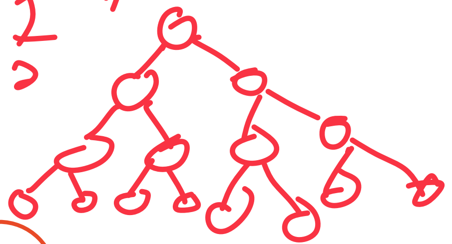
Print: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

Pre order!

Breadth First Search

size of queue ~ width of tree

height width
1 2
2 4
3 8
4 16



Fully explore depth i before exploring depth i+1

Make a queue initialized with root

While queue isn't empty:

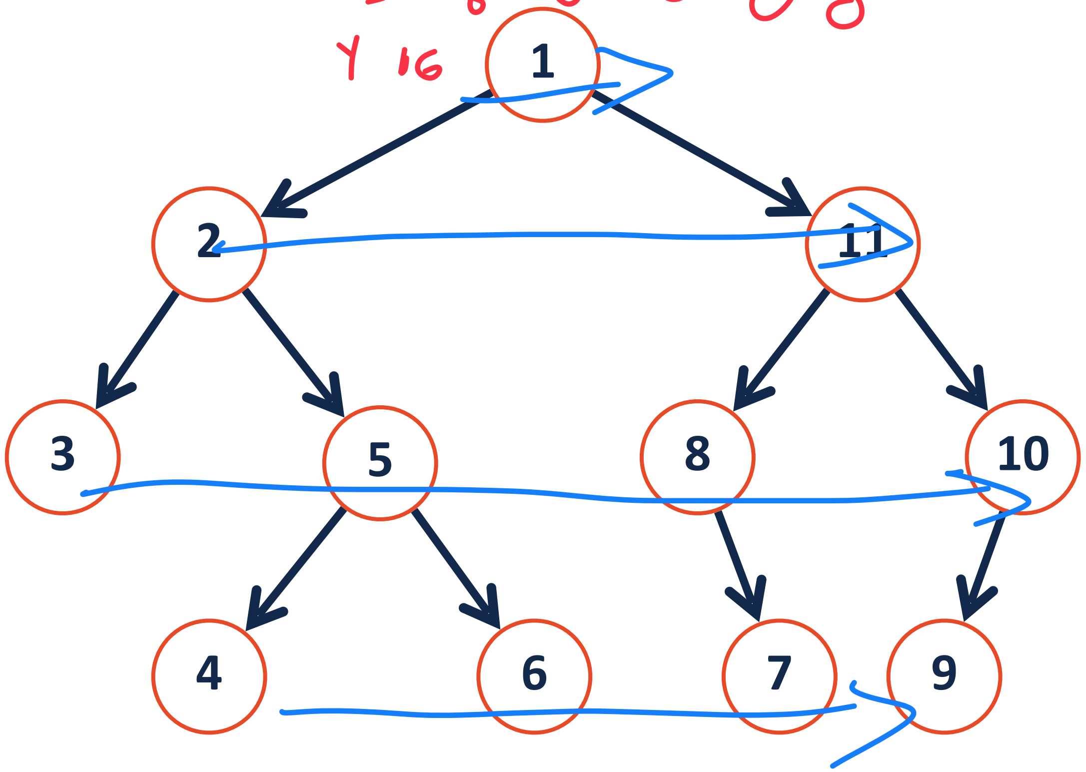
Dequeue front element (as tmp)

Print tmp

Enqueue tmp->left

Enqueue tmp->right

FIFO



equal to width

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

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

Breadth First Search

Fully explore depth i before exploring depth i+1

Make a queue initialized with root

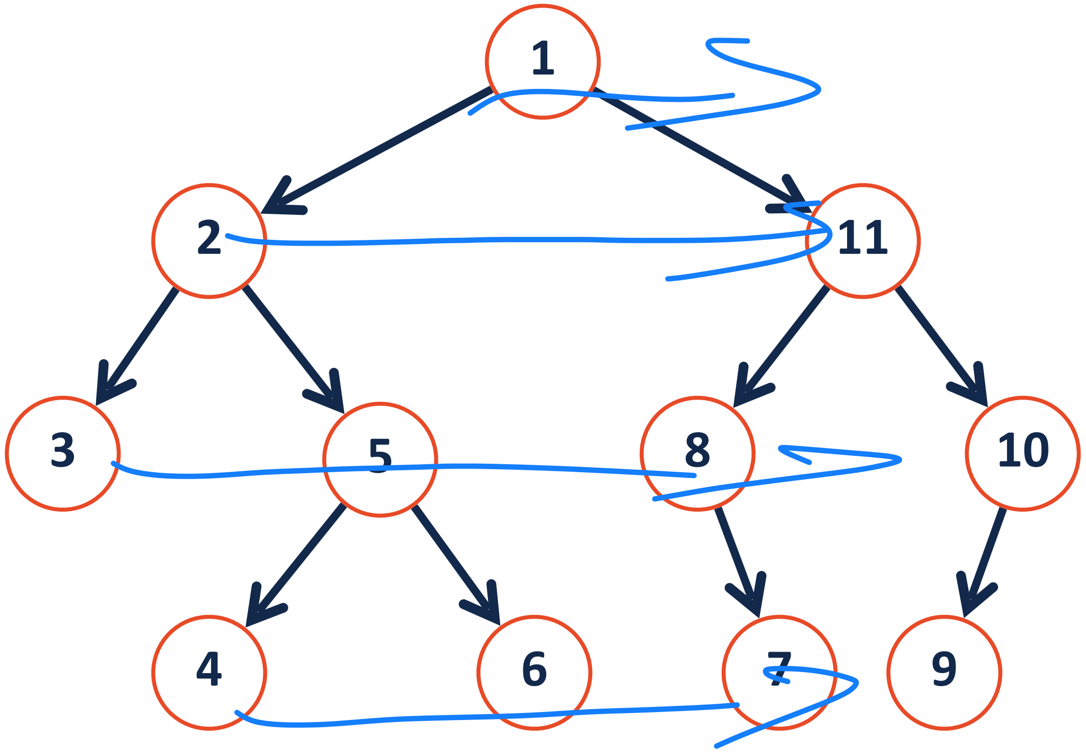
While queue isn't empty:

  Dequeue front element (as `tmp`)

  Print `tmp`

  Enqueue `tmp->left`

  Enqueue `tmp->right`



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

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

BST Find

find(66)

root → nullptr

root → 66

Start @ root

Recursive Problem!

Base Case:

↳ If tree empty, return root

↳ If root is query, return root

Recursive step:

Compare $\underbrace{\text{root→key}}_{\text{tmp}}$ w/ query

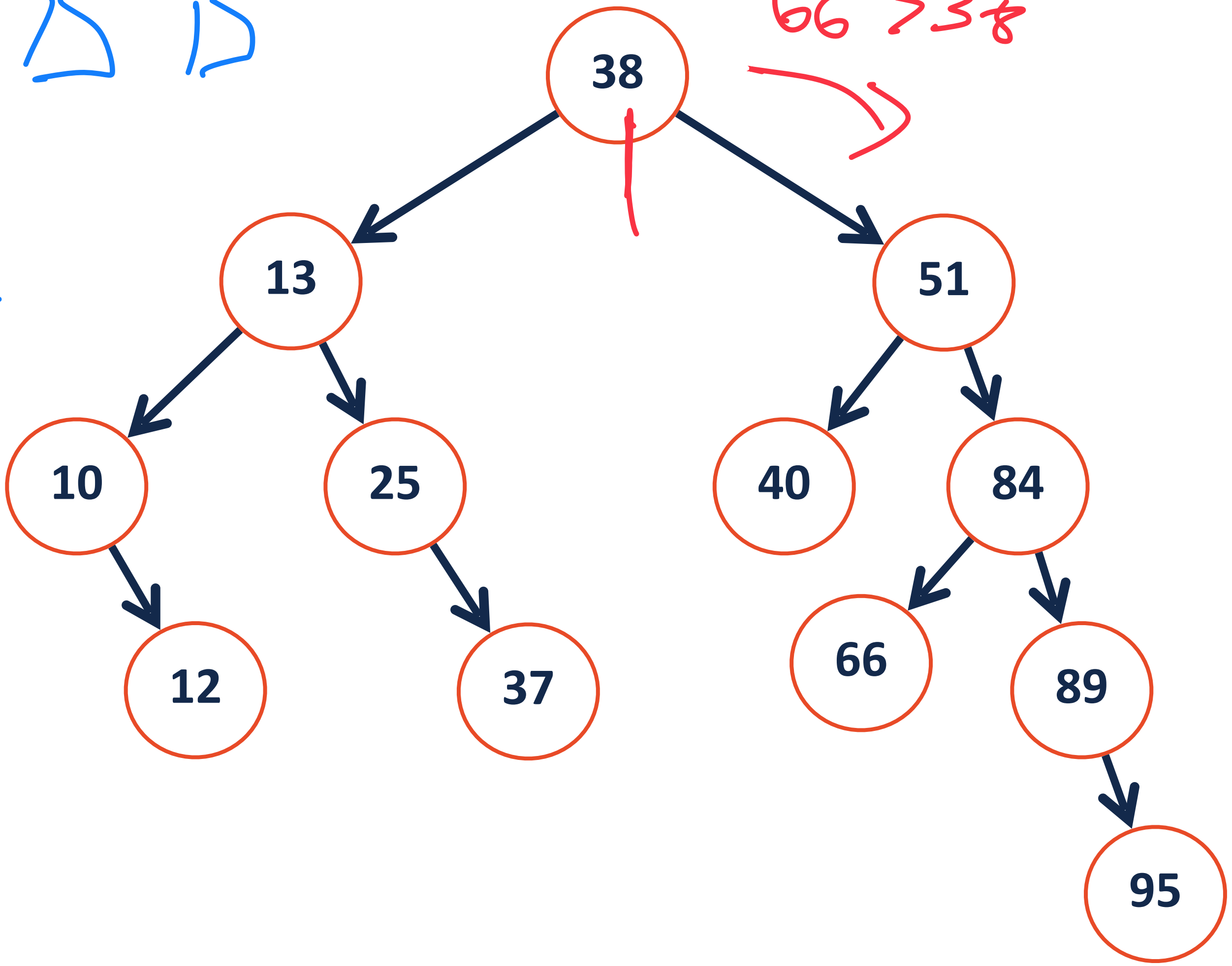
if

tmp > query, recurse right

tmp < query, recurse left

==

66 > 38



BST Find

`find(66)`

A recursive function based around value of root:

Base Case: If root is `null`, return root

Let `tmp = root->key()`

`tmp == query`, return root

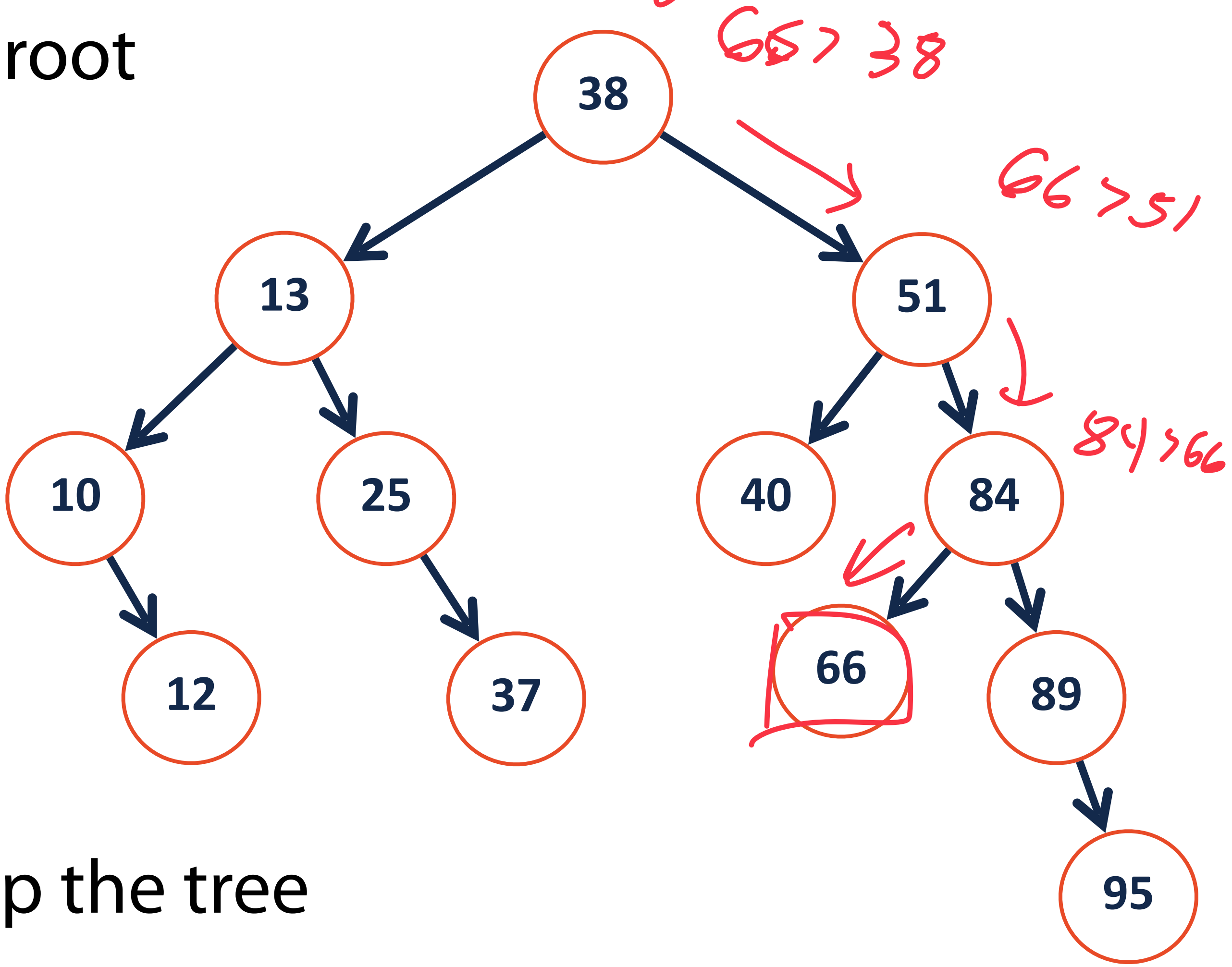
Recursion:

`tmp <  query`, recurse right

`tmp >  query`, recurse left

Combining:

Return the recursive value back up the tree



query

No const here

```
template<typename K, typename V>

______TreeNode *&________ _find(TreeNode *& root, const K & key) {

// Base Case
if(root == nullptr || root->key == key){
    return root;
}

// Recursive Step ("Combining step" is 'return')
if (root->key > key){
    return _find(root->left, key);
}

return _find(root->right, key);


}
```

No const here

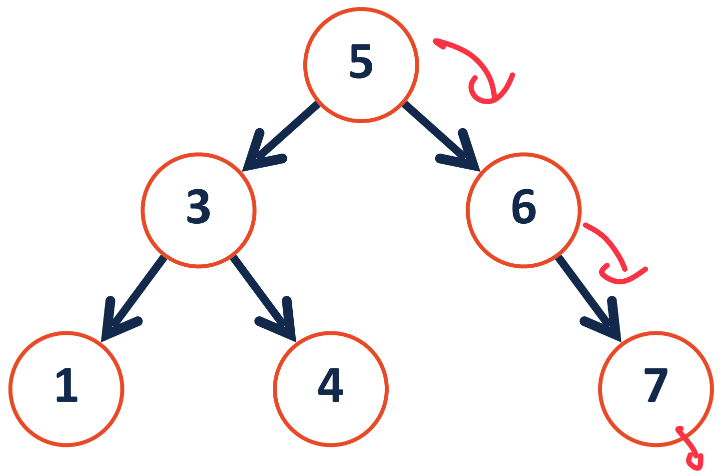
Find(8) returns pointer by ref (7->right)

Not nullptr!

Smaller go left

"else"

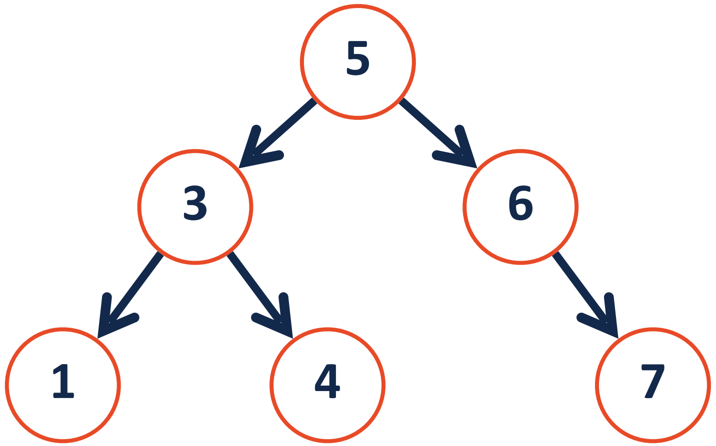
larger go right



```
template<typename K, typename V>

void _insert(const K & key, const V & val) {

    return _insert(root, key, val);
}

```

```
template<typename K, typename V>

void _insert(TreeNode *& root, const K & key, const V & val) {

TreeNode *& tmp = _find(root, key);


tmp = new treeNode(key, val);




}



```

find is key!



```
template<typename K, typename V>

void _remove(TreeNode *& root, const K & key) {
```

This weeks lab!

0 - child

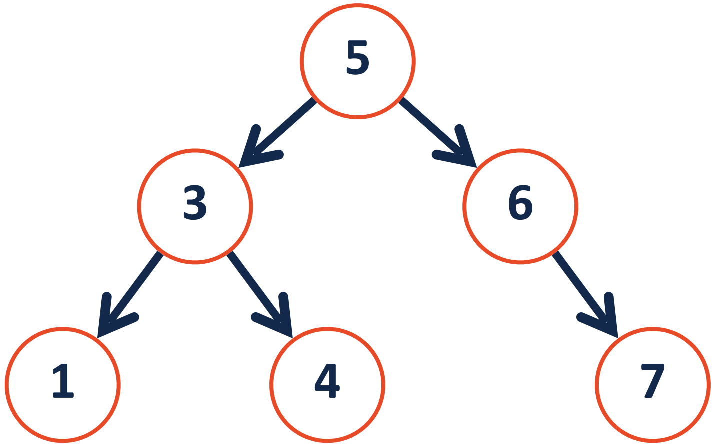
1 - child

2 - child → find IOP/IOS

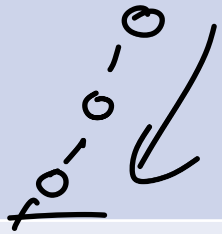
↳ swap IOP w/ target

↳ remove target

```
}
```



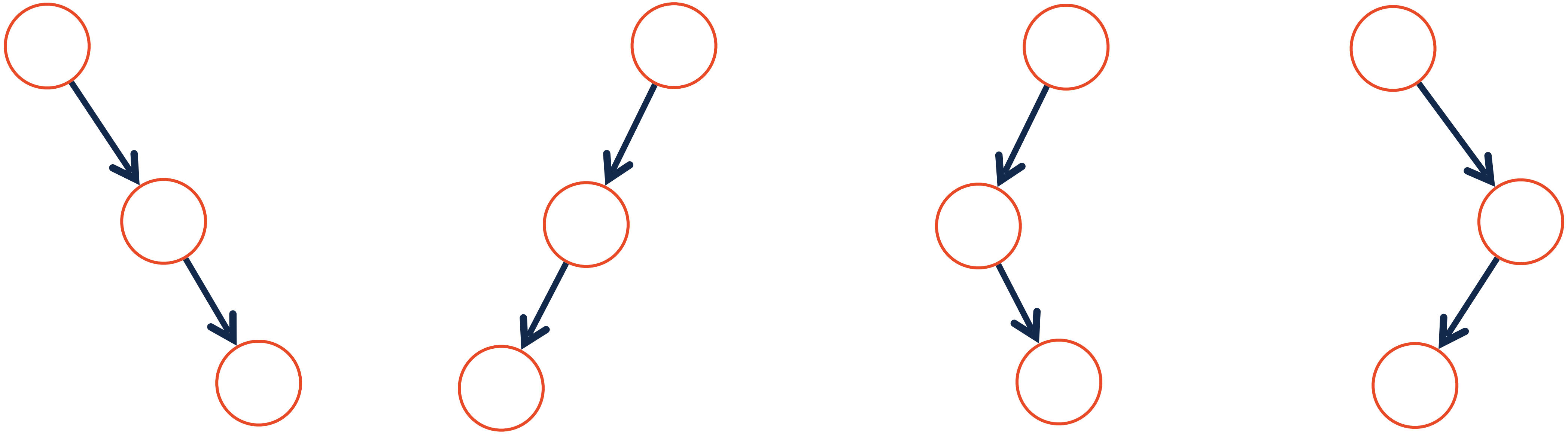
BST Analysis – Running Time

Operation	BST Worst Case
find	$O(h)$ 
insert	$O(h)$ b/c
remove	$\underbrace{O(h)}_{\text{find}} + \underbrace{O(h)}_{\text{find (IOP)}} + \underbrace{O(h)}_{\text{remove()}} = O(h)$
traverse	$O(n)$

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)

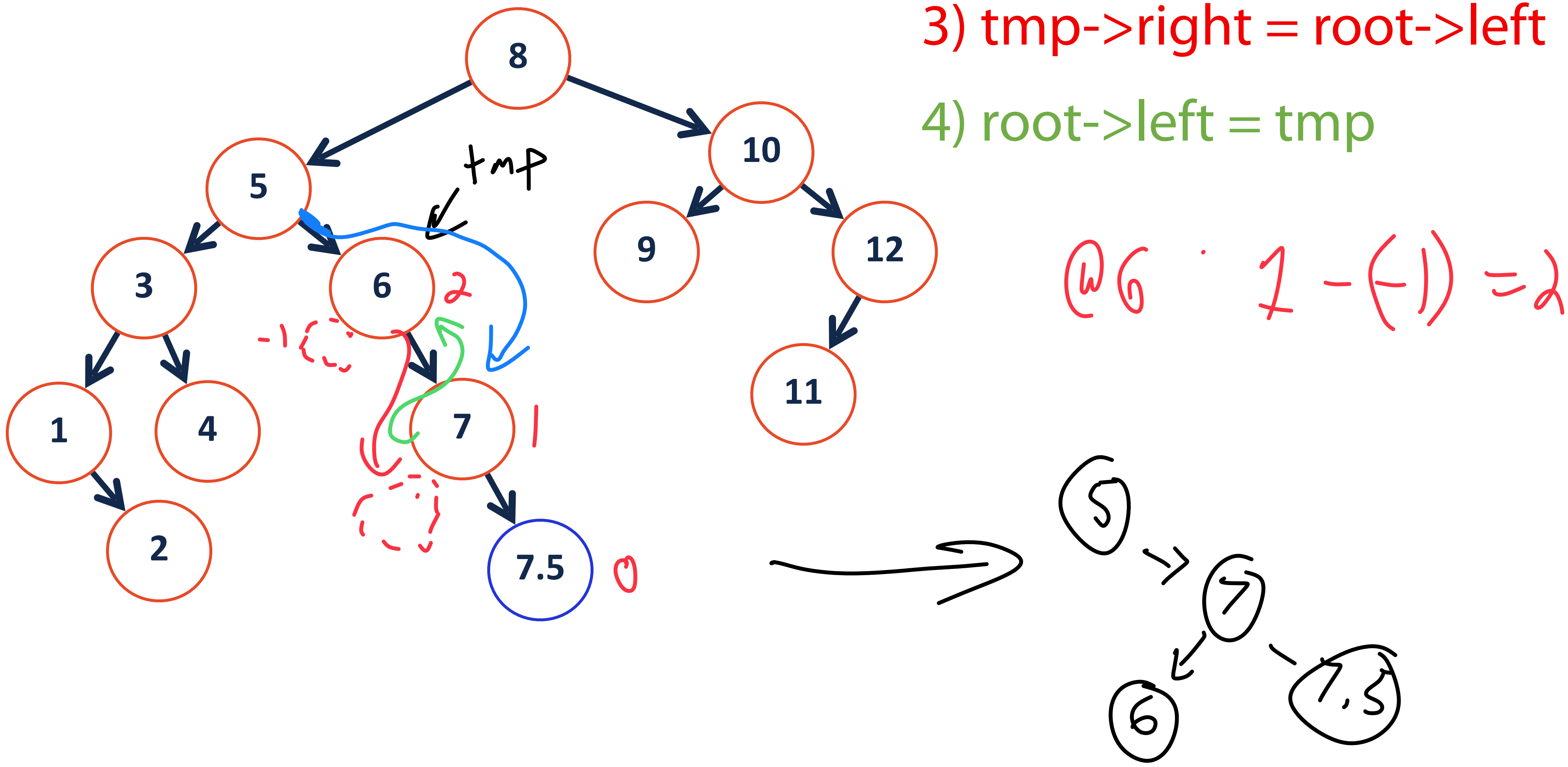
AVL Rotations



	Left	Right	LeftRight	RightLeft
Root Balance:	2	-2	-2	2
Child Balance:	1	-1	1	-1

Left Rotation

1) Create a tmp pointer to root
2) Update root to point to mid
3) tmp->right = root->left
4) root->left = tmp



AVL Rotations

Four kinds of rotations: (L, R, LR, RL)

1. All rotations are local (subtrees are not impacted)

2. The running time of rotations are constant

3. The rotations maintain BST property

Goal: AVL tree will be balanced

↳ This will make height bounded by log(n)

AVL Tree Analysis

For an AVL tree of height h:

Find runs in: $O(h)$.

Insert runs in: $O(h)$.

Remove runs in: $O(h)$.

Guarantee:

1) Tree is balanced

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

Nearest Neighbor: k-d tree

Find medians in all dim

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

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

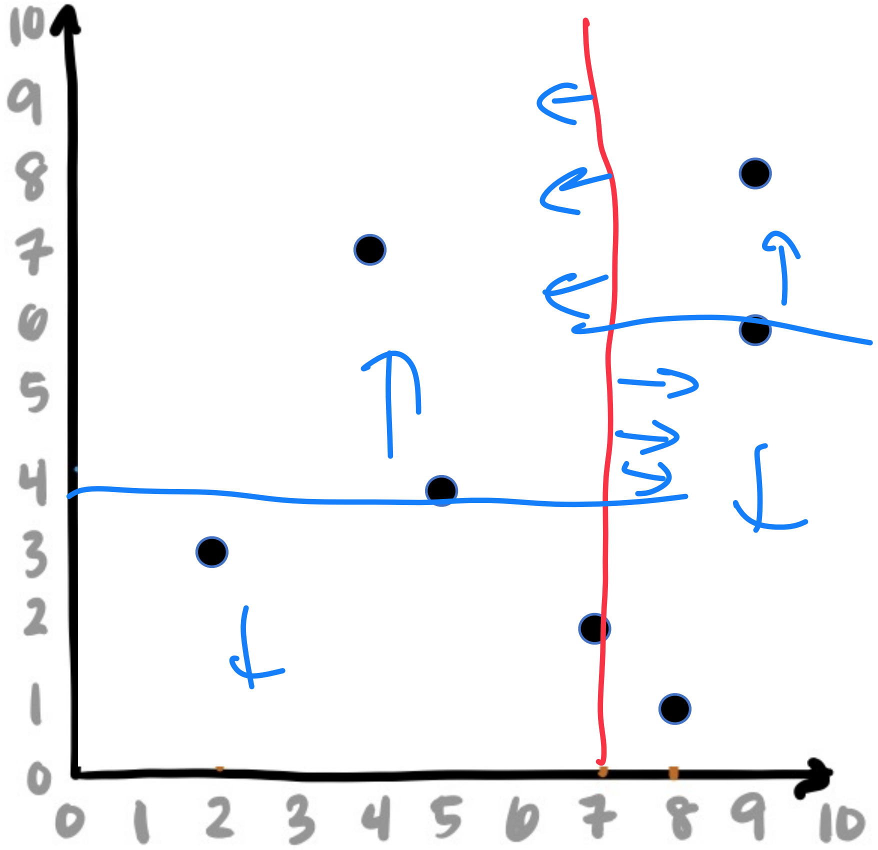
Median of all items in x dim

(7,2)

x

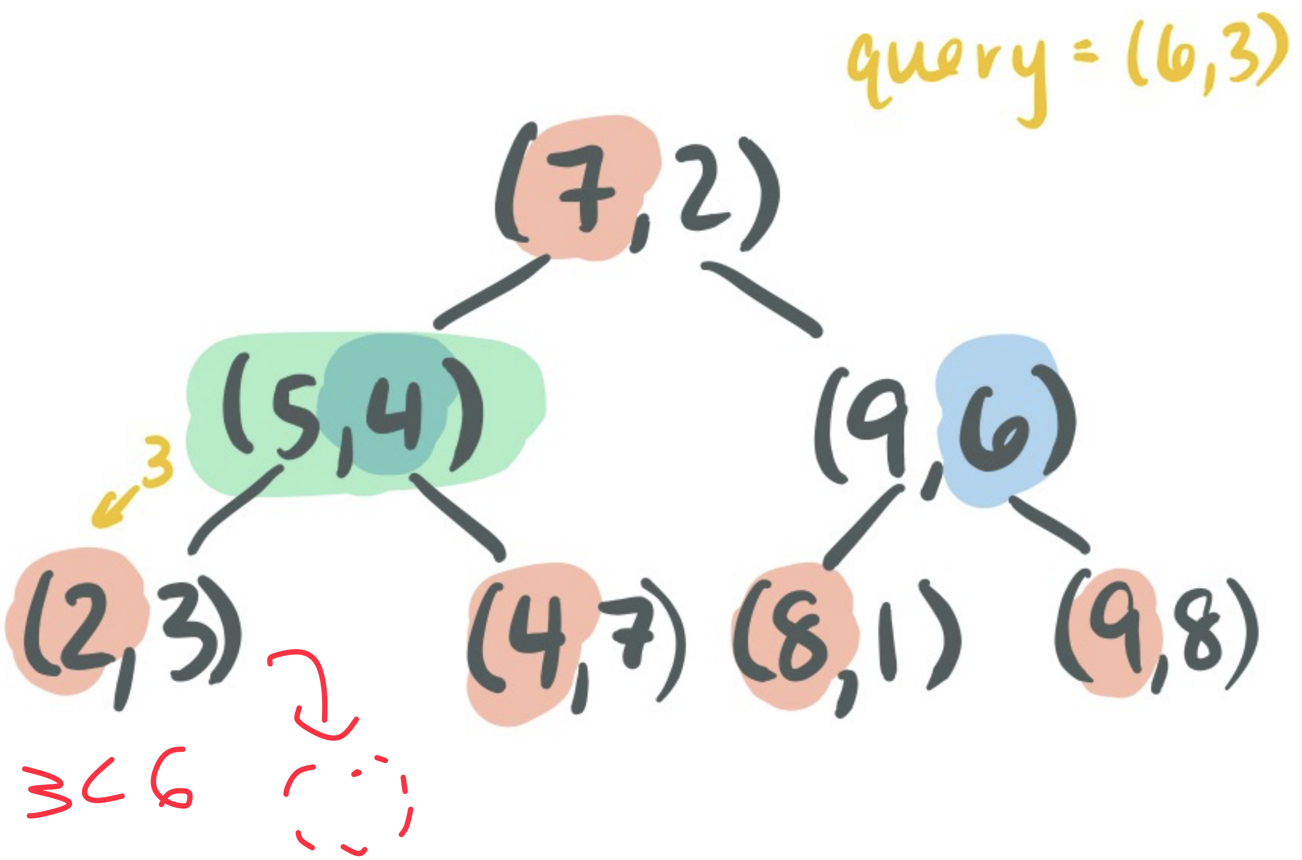
(5,4) (9,6)

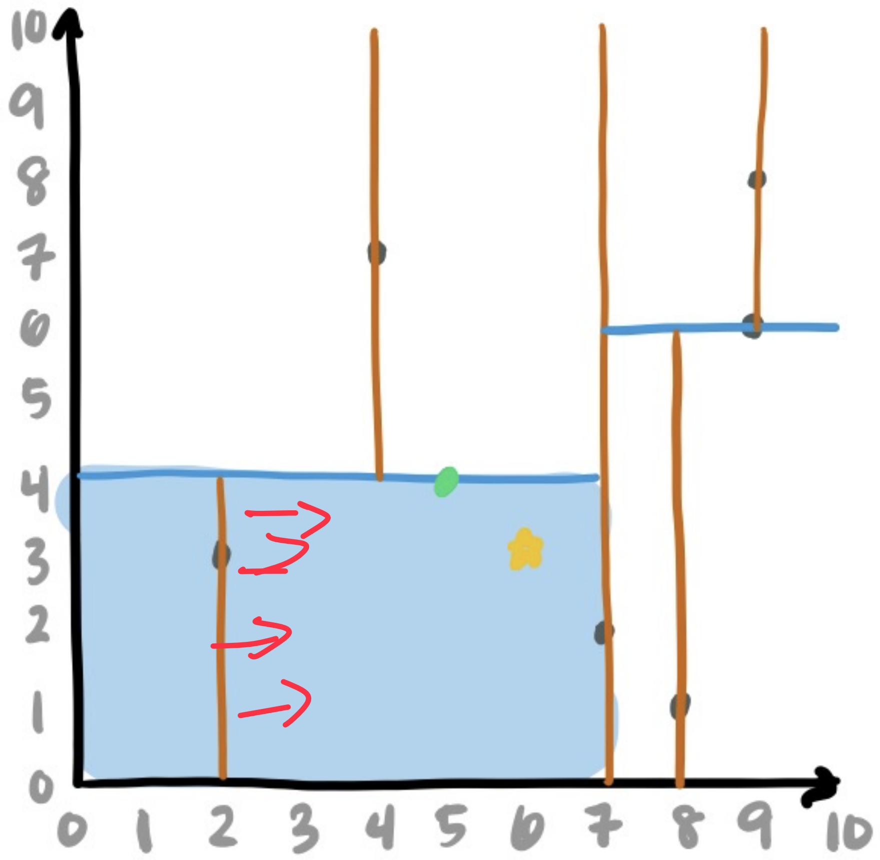
med of all items x<7 in the y dim



Nearest Neighbor: k-d tree

Search by comparing query and node in single **alternating** dimension

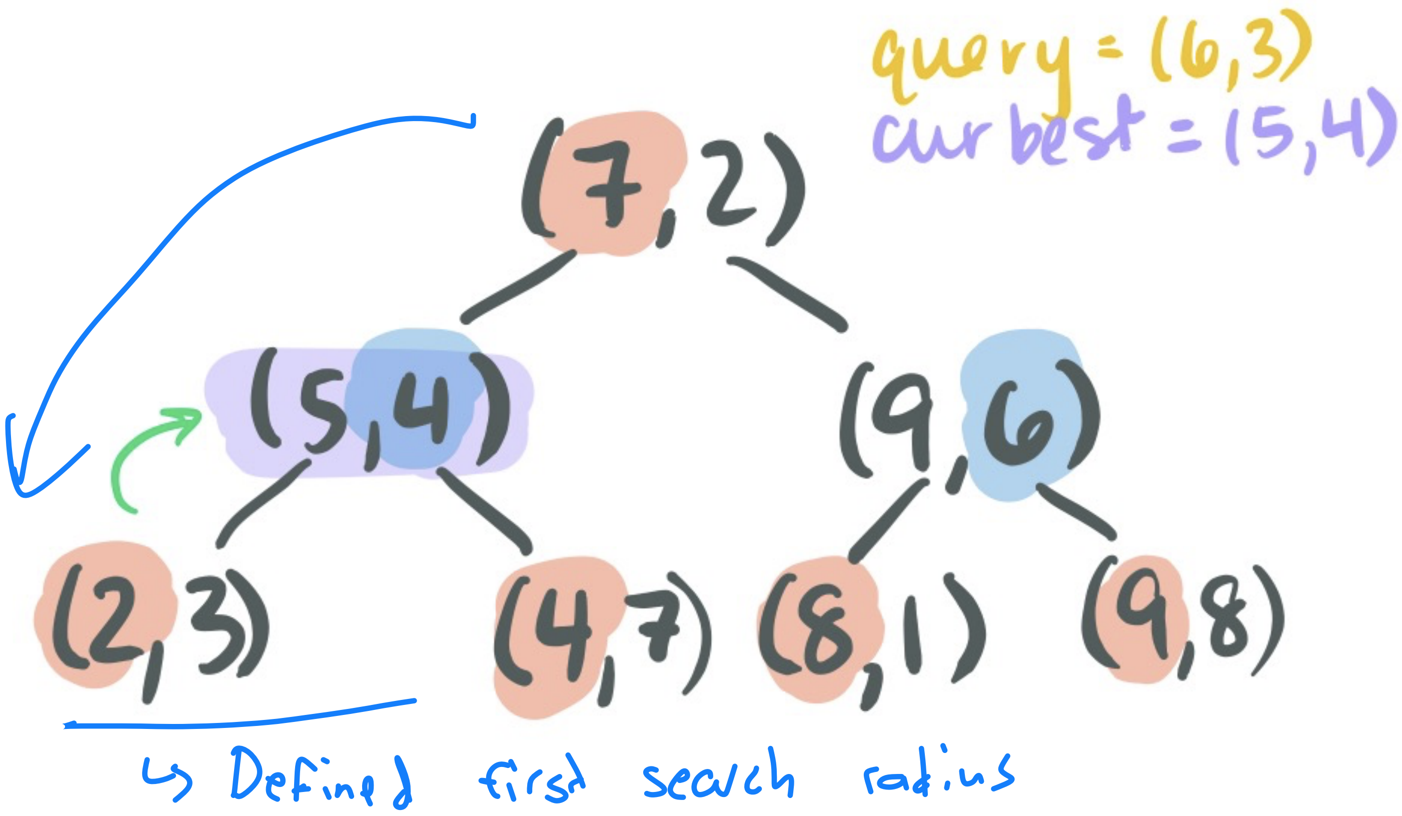


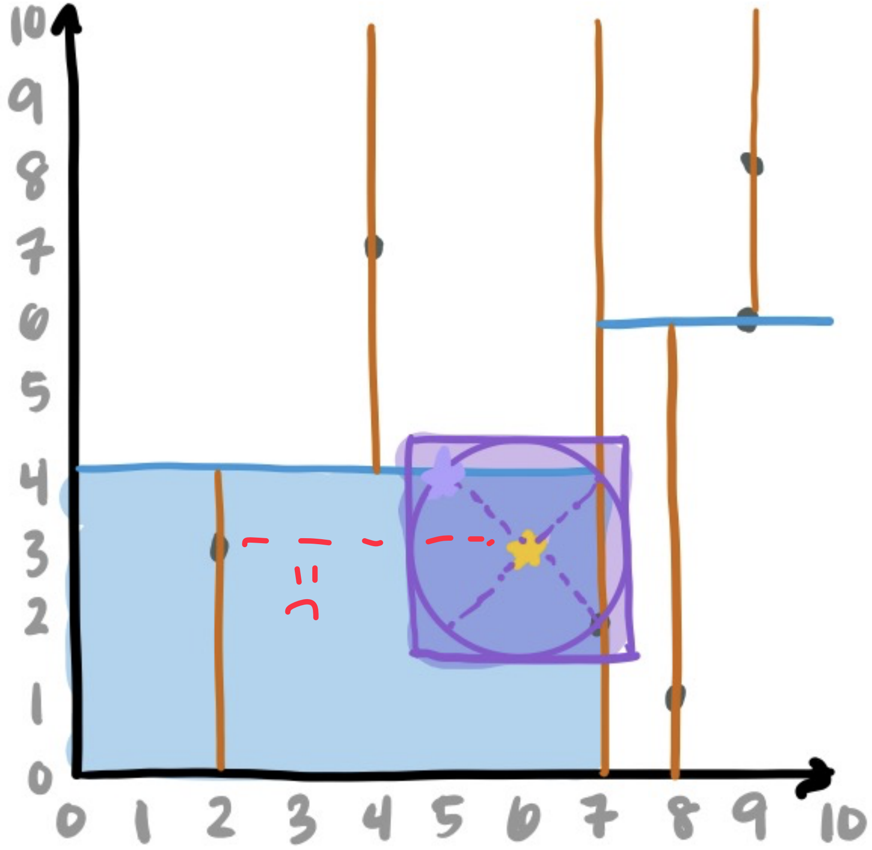


Nearest Neighbor: k-d tree

Backtracking: start recursing backwards -- store "best" possibility as you trace back

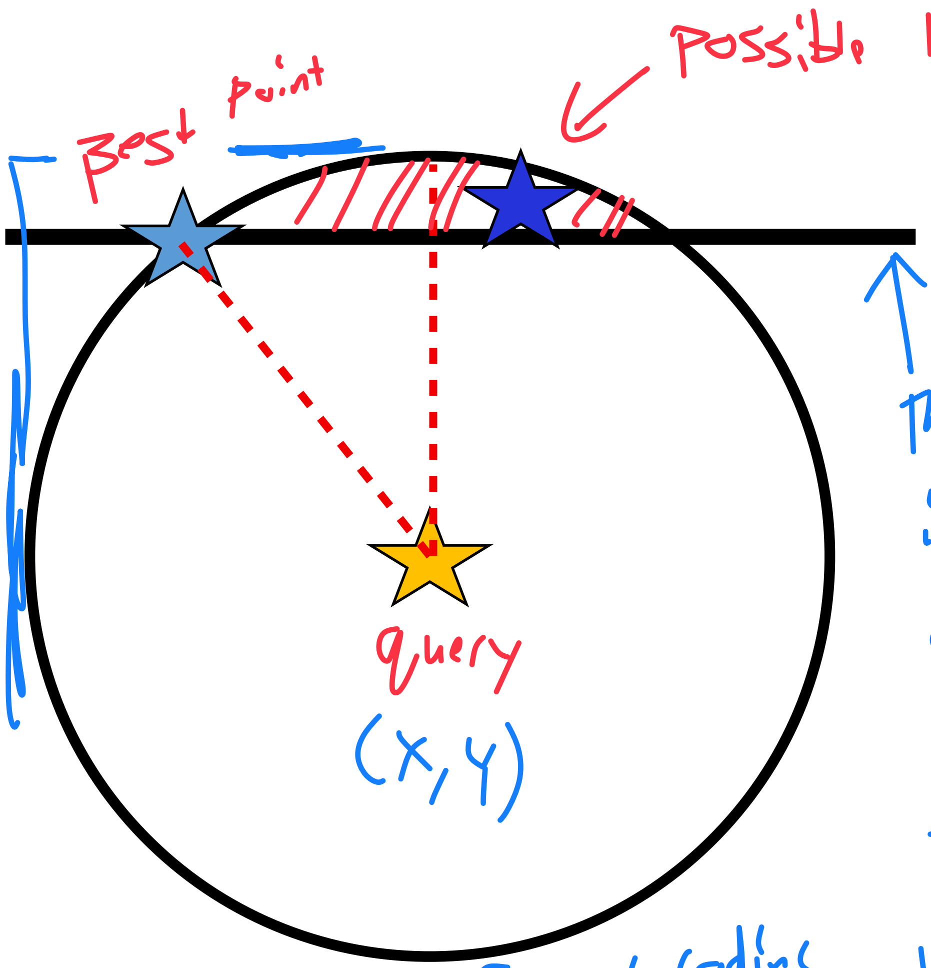
(2,3) or (5,4) better nearest point?





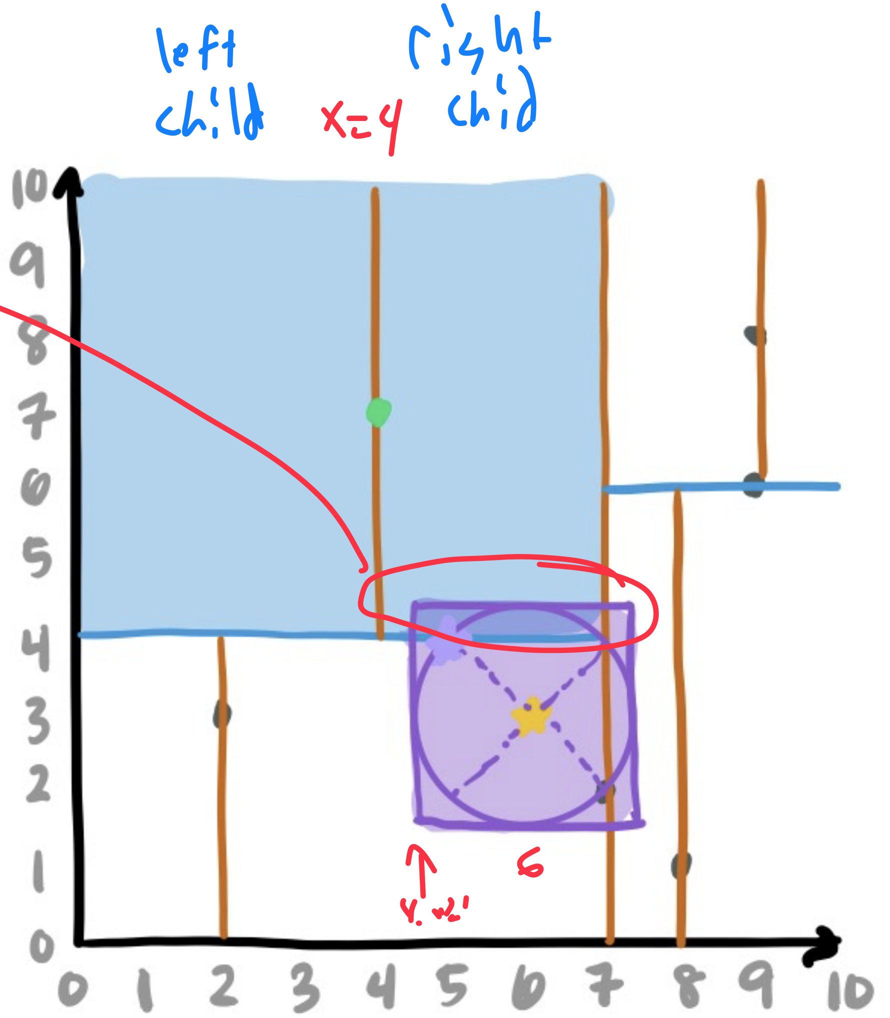
Nearest Neighbor: k-d tree

May have to recursively check other branches of tree — **why?**

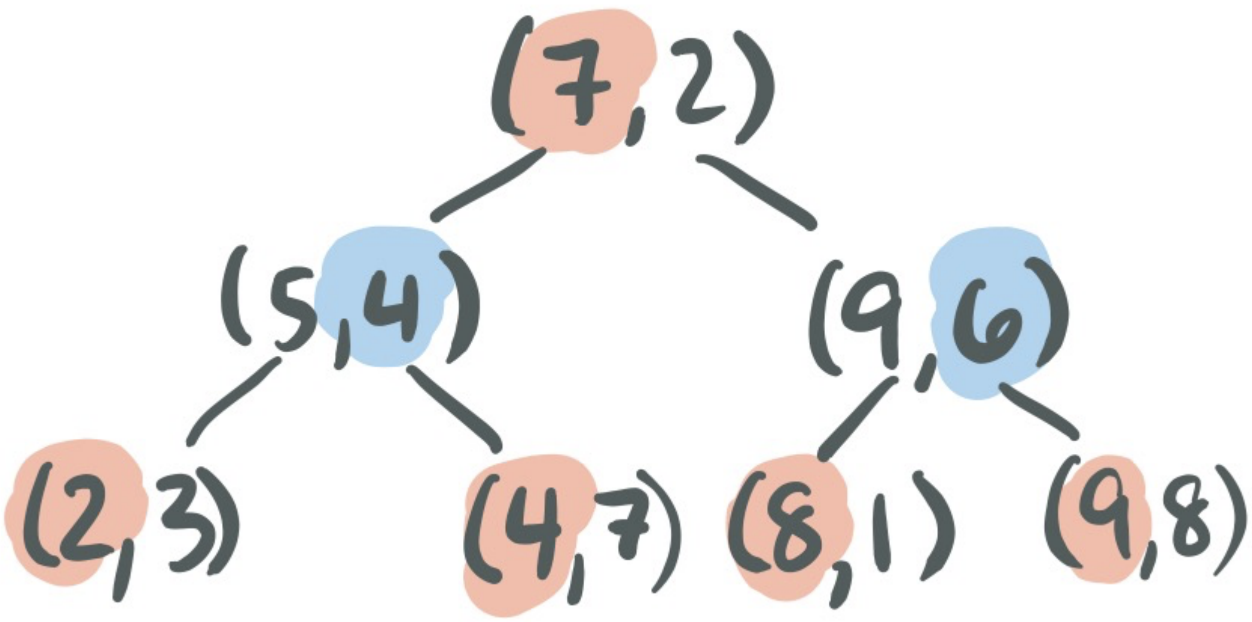


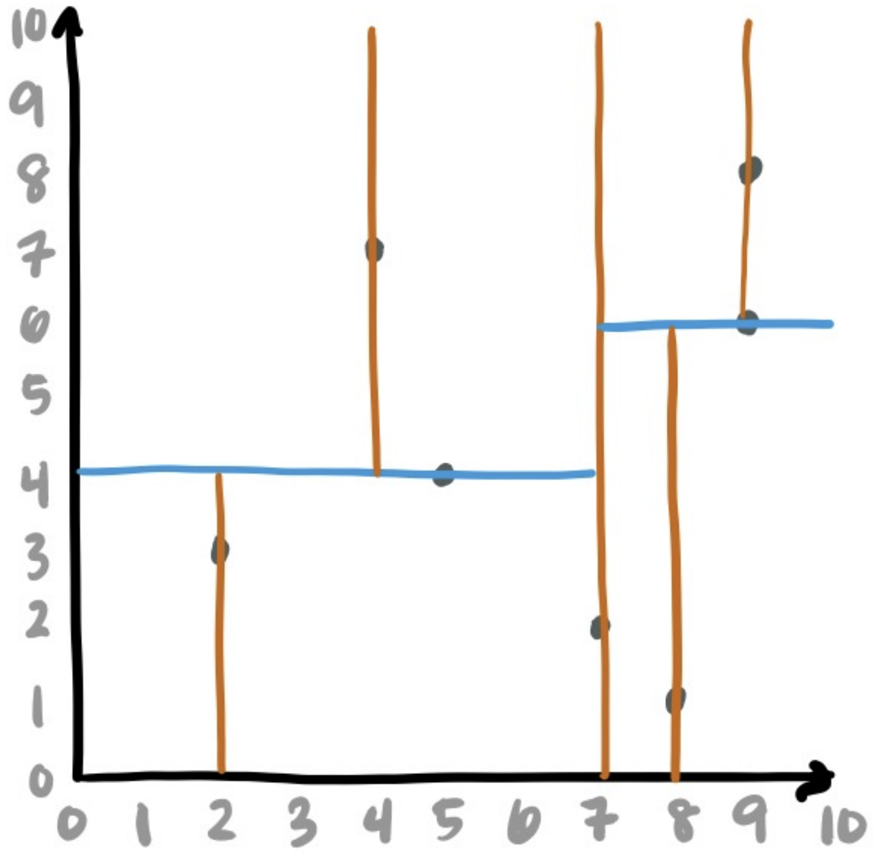
This line is important for deciding overlap

Qy + radius (dist) > vs y=4 line



Nearest Neighbor: k-d 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**

→ 0 children

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

If int$\left(\frac{m}{2}\right)$ is keys, +1 is children

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

All leaves in the tree are at the same level.

BTree Find

Note edge case if no larger value

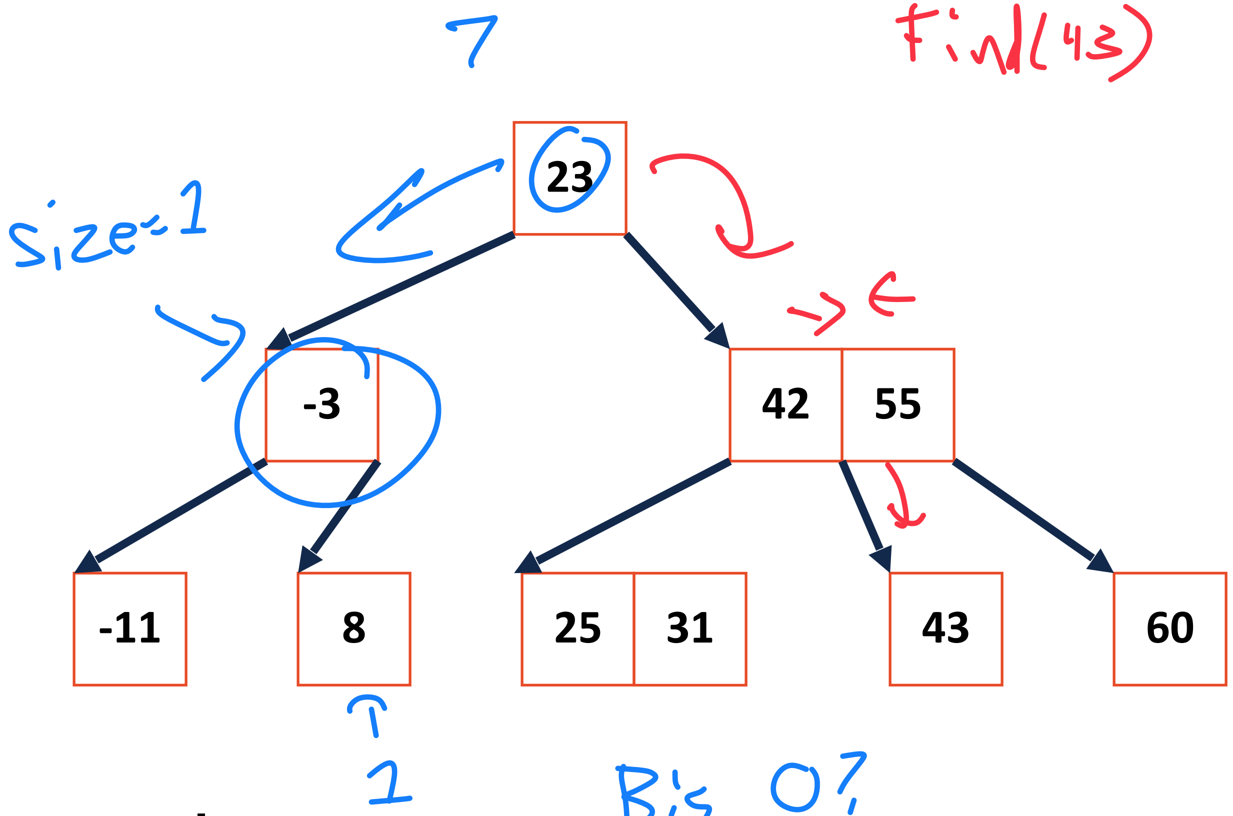
Find(7)

Find(43)

Base Case:

If root is empty, return

If leaf, do array find() and return



Recursive Step:

Array find() for match or first greater value

Recurse on appropriate child *

B is 0?

O(m) x Nodes

↳ O(log n)

this is constant

Tip: Index of first greater value is index of child we want to visit!

BTree Insertion

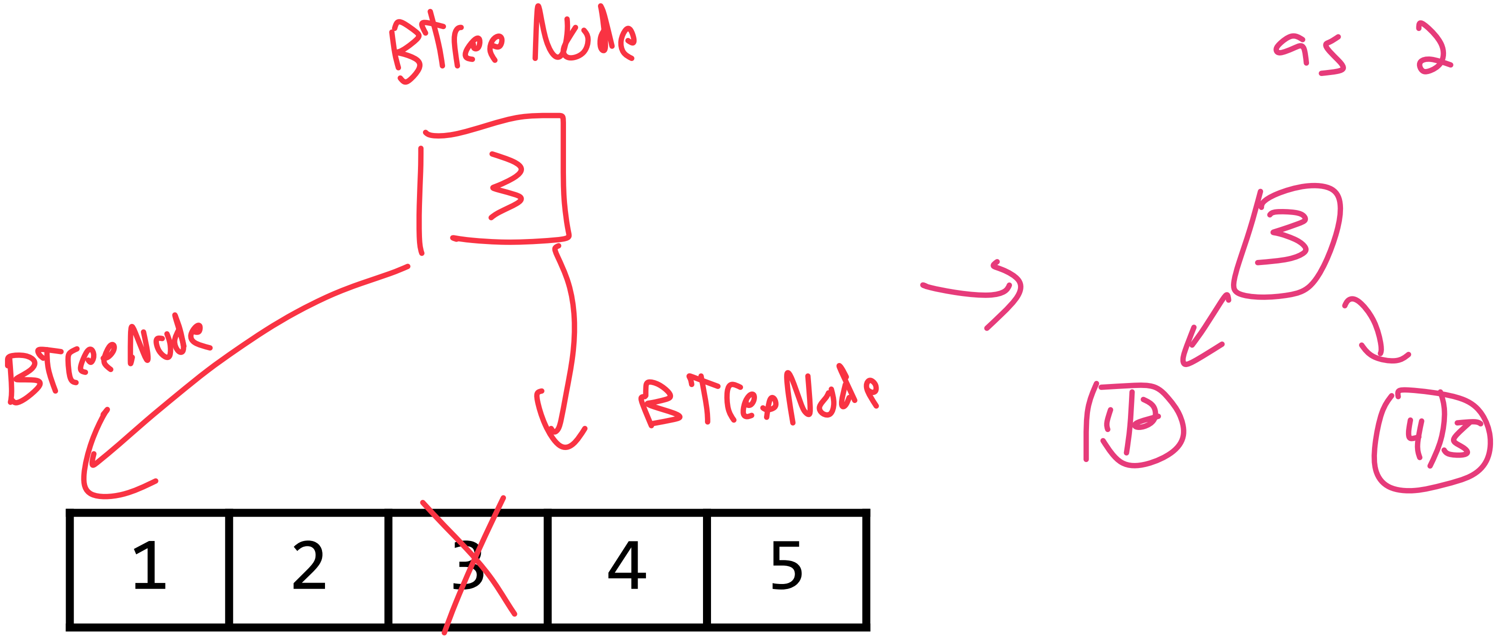
M = 5

When we hit **M** items, split into three nodes!

1) Find median
2) "Raise median up"

↳ Cut array in half as 2 new BTree Nodes

Insert(1)
Insert(2)
Insert(3)
Insert(4)
Insert(5)
Insert(6)
Insert(7)
Insert(8)

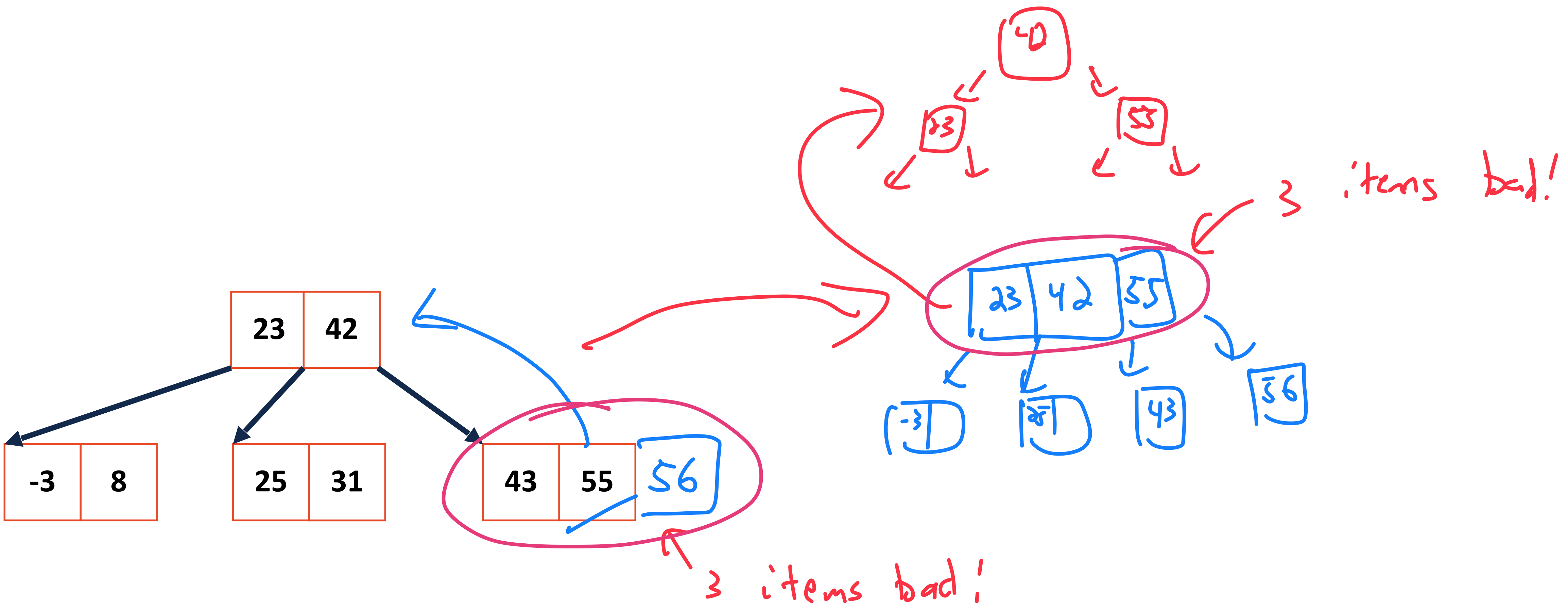


M - 1 items MAX

BTree Recursive Insert

`Insert(56), M = 3`

Insert always starts at a leaf but can propagate up repeatedly.

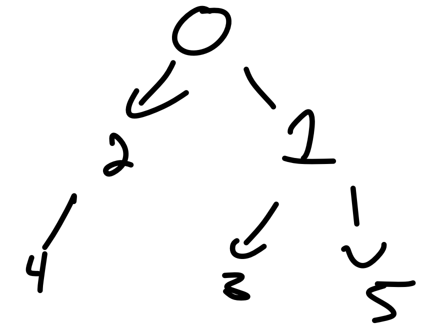


Final thoughts on Trees

Trees have a large space of **possible coding questions**

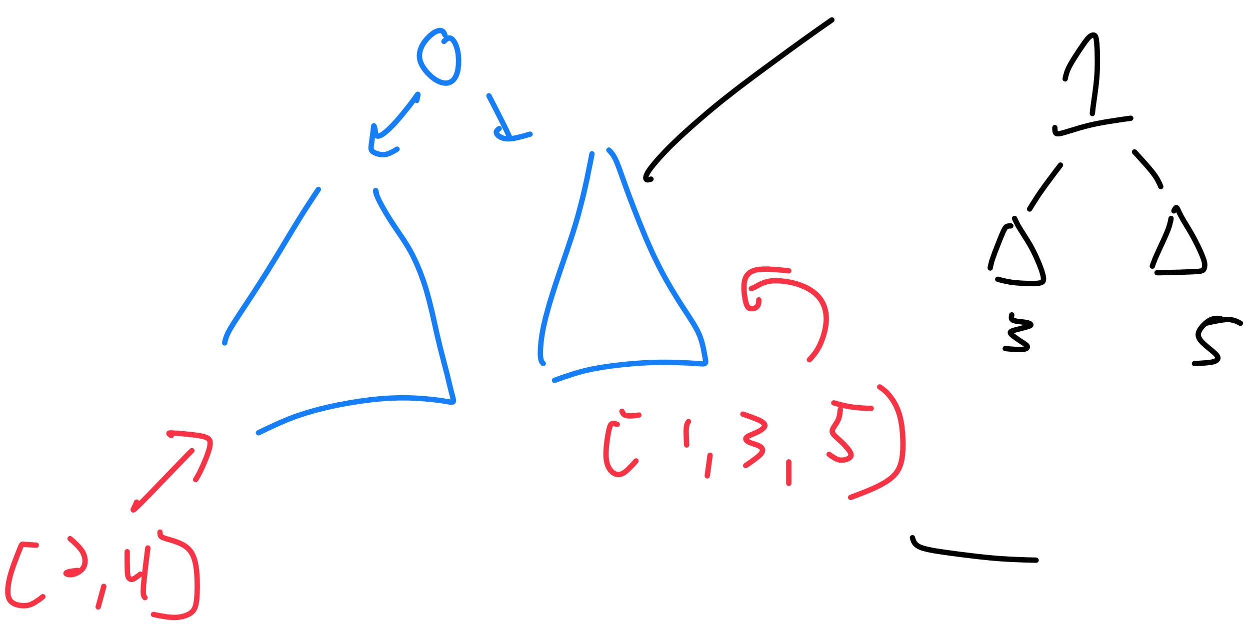
We hit **tree iterators** multiple times…

You saw **tree constructors of unusual shapes**…



[0, 1, 2, 3, 4, 5]

even indices left
odd indices right



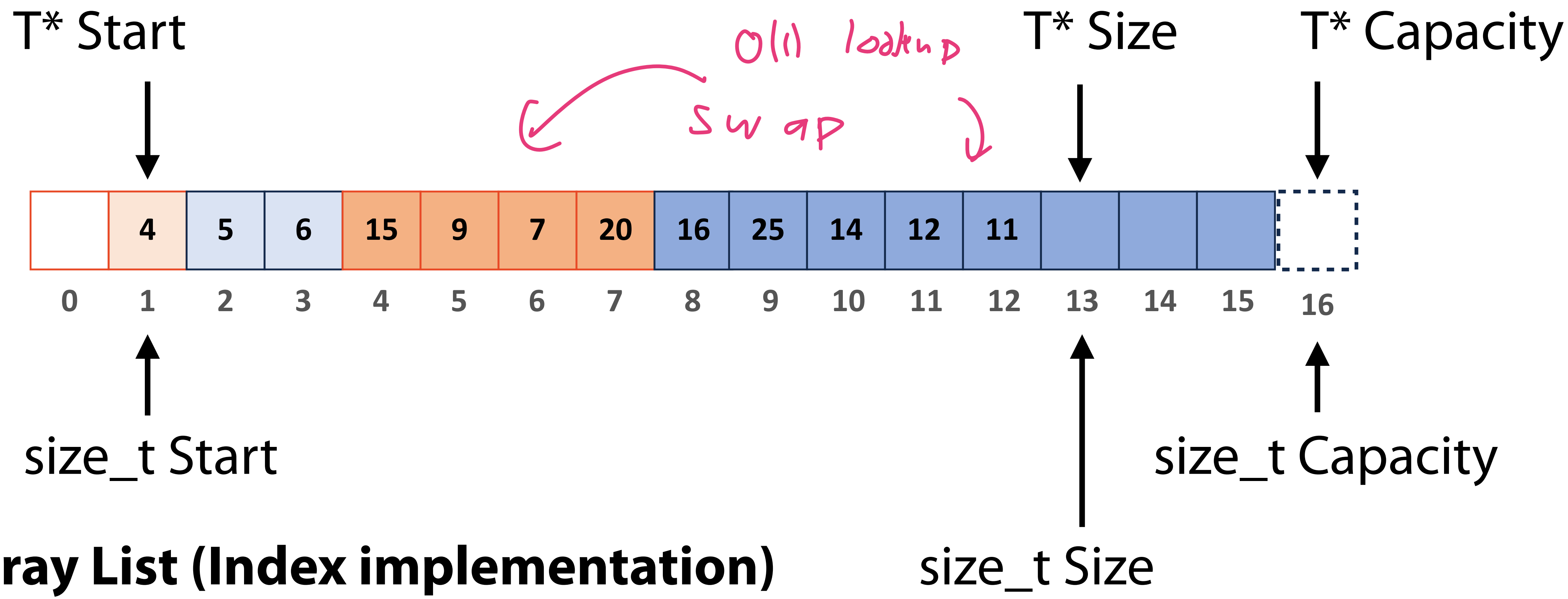
Heap

Taking advantage of special cases in lists / arrays

insert Back

O(1)*

Array List (Pointer implementation)



Array List (Index implementation)

(min)Heap (Priority Queue)

By storing as a complete tree, can avoid using pointers at all!

$2^{h+1} - 1$

← n extra space

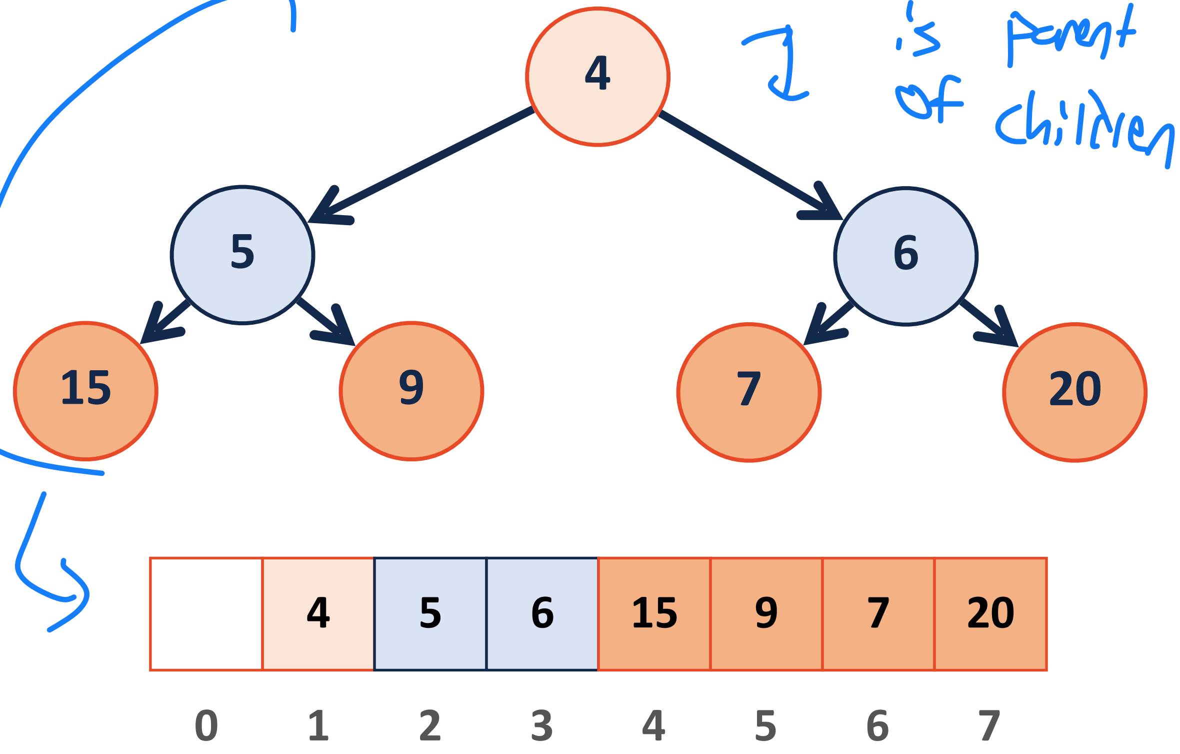
If index starts at 1:

`leftChild(i): 2i`

`rightChild(i): 2i+1`

`parent(i): floor(i/2)`

min value is parent of children



insert

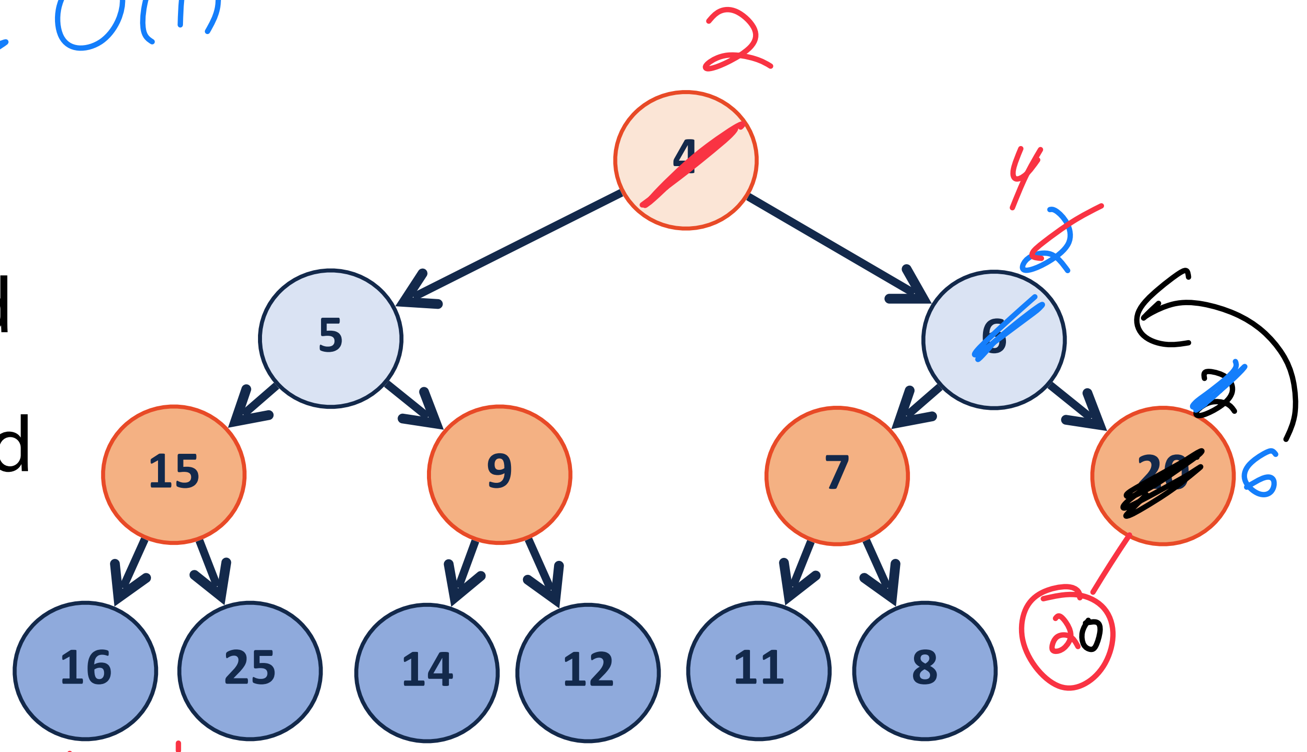
Insert(2)

1) Insert at end of array $O(1)^*$

2) Check if minHeap still valid

3) Swap with parent if needed $O(1)$

Steps 2 and 3 are recursive!



$\lfloor \frac{i}{2} \rfloor$

$P(i) = i/2$

$i = 7$

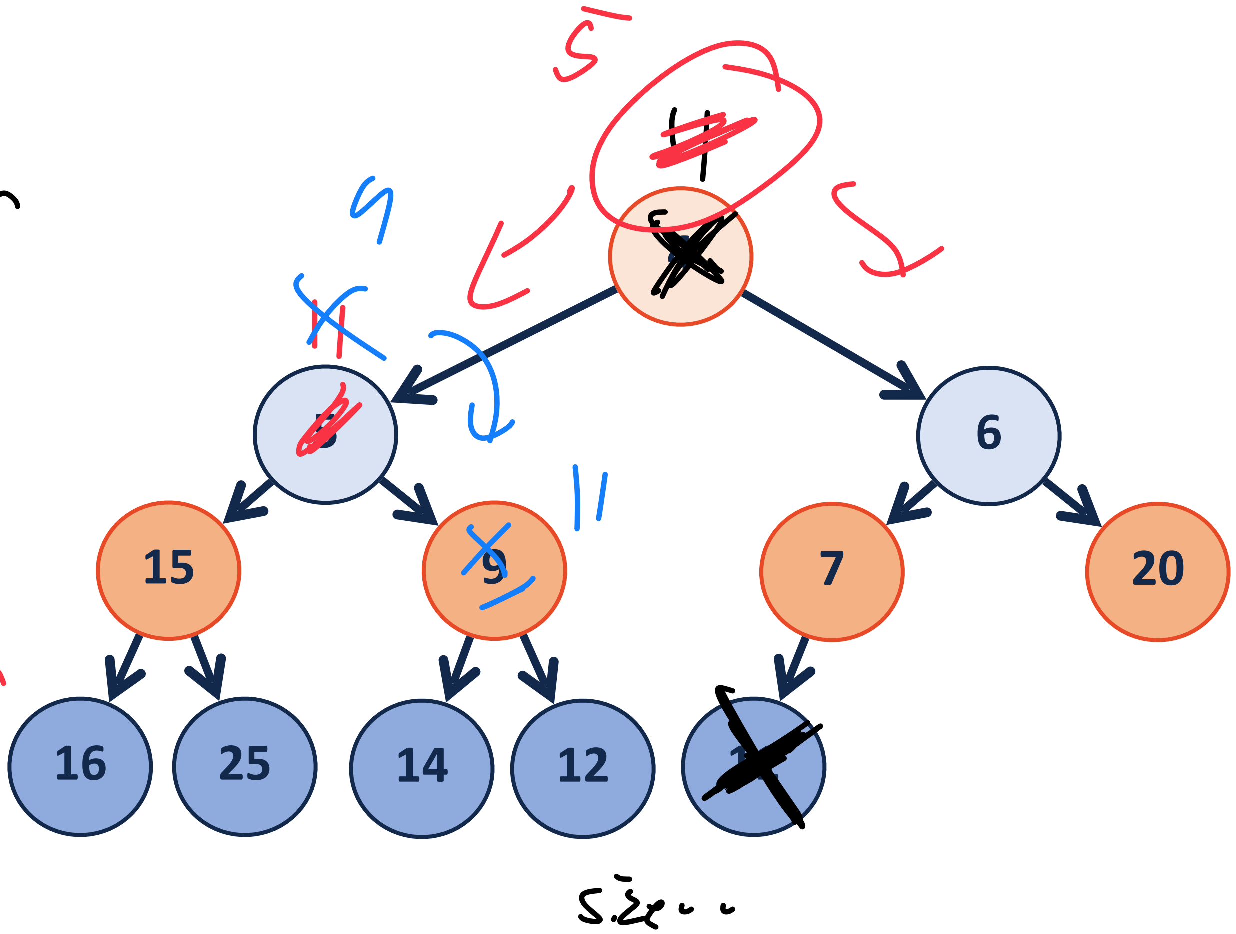
i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	~~4~~ 2	5	~~6~~	15	9	7	~~20~~	16	25	14	12	11	8	~~2~~	
			~~2~~ 4				~~2~~ 6							20	

removeMin

1) Swap root w/ last item
   ↳ Delete last item
   ↳ size--;

2) heapifyDown()
   ↳ Repeated swaps w/ min child
   until leaf or smaller than both children



	11											4			
	~~4~~	5	6	15	9	7	20	16	25	14	12	~~11~~			
	↑ 1											↑ size - 1			

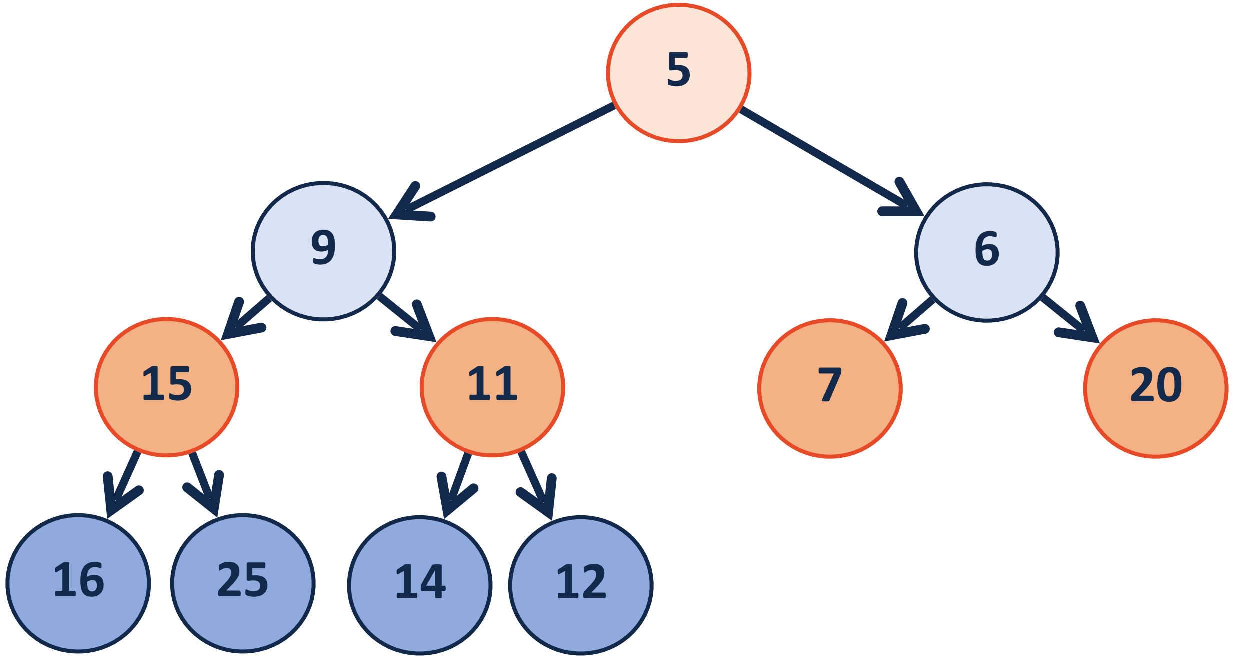
removeMin

1) Swap root with last item

    (and remove)

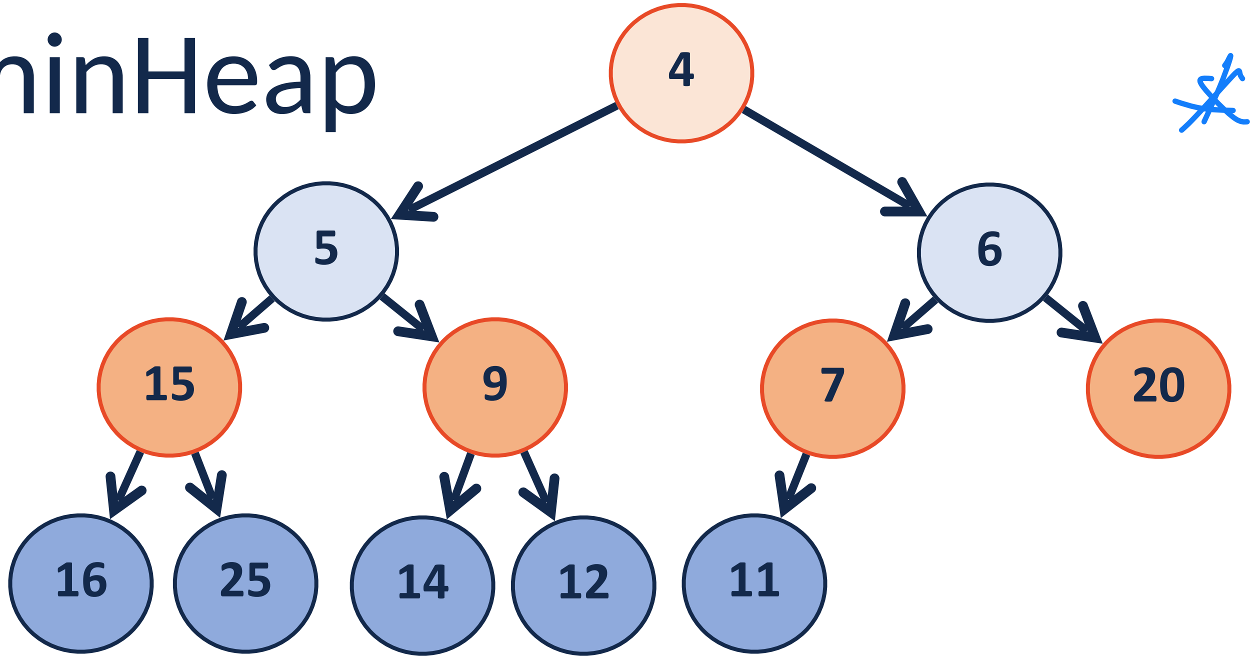
    (and modify size)

2) HeapifyDown( ) root



	5	9	6	15	11	7	20	16	25	14	12				

minHeap



1. **Construction**
   ↳ $O(n)$
2. **Insert** → $O(\log n)$
3. **RemoveMin** → $O(\log n)$

On an array!

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

minHeap is a good example of tradeoffs:

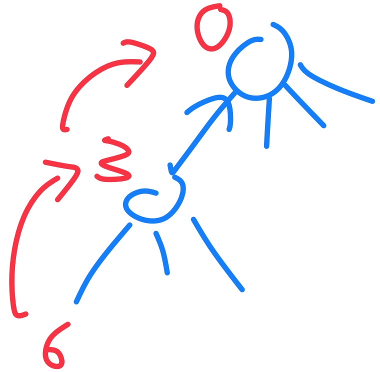
Array is faster than tree (memory)

+ Improved construction

↪ No random access ??

Disjoint Sets

Disjoint Set Implementation



Taking advantage of array lookup operations

Store an UpTree as an array, canonical items store **height** / **size**

0 1 4

2 7

3 5 6

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

Find(k): Repeatedly look up values until **negative value**

Union(k_1, k_2): Update ***smaller*** canonical item to point to larger

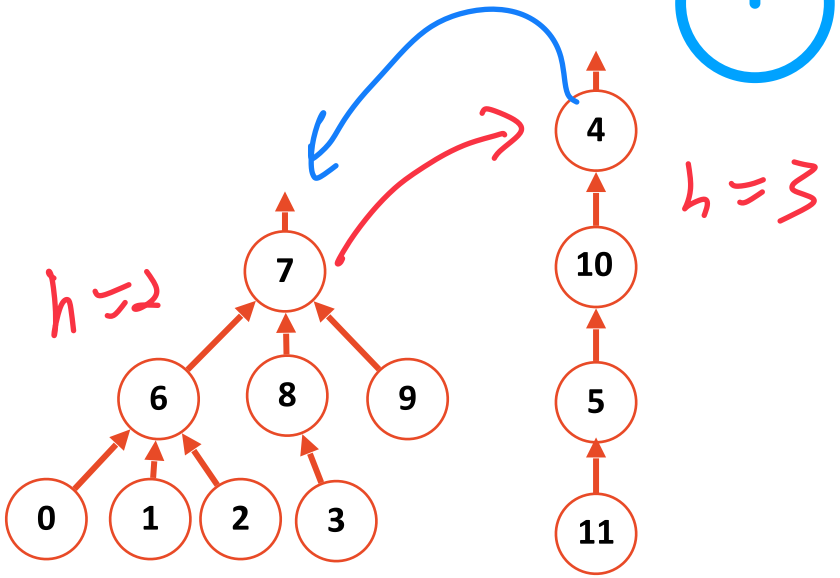
Update value of remaining canonical item

O(1)

Disjoint Sets – Smart Union

Two O(1) methods of combining two sets

Claim: Both limit height to: O(log n).



Union by height

Before Union

4	…	7
-4		-3

After Union

4	…	7
-4		4

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

Union by size

Before Union

4	…	7
-4		-8

After Union

4	…	7
7		-12

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

Disjoint Sets Path Compression

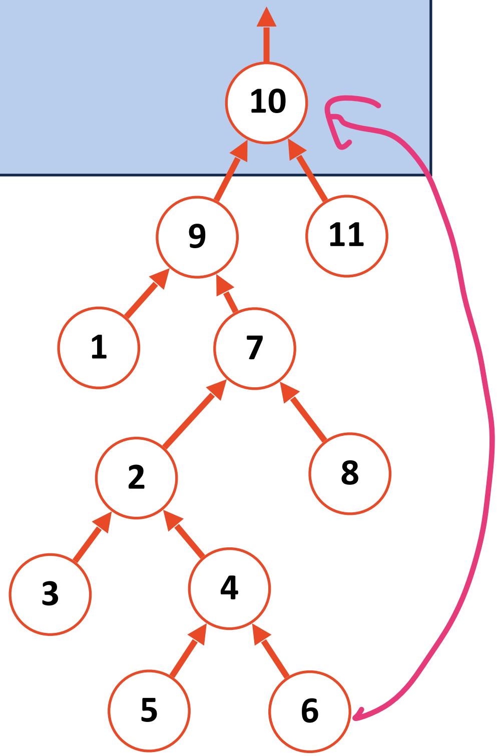
Find(6)

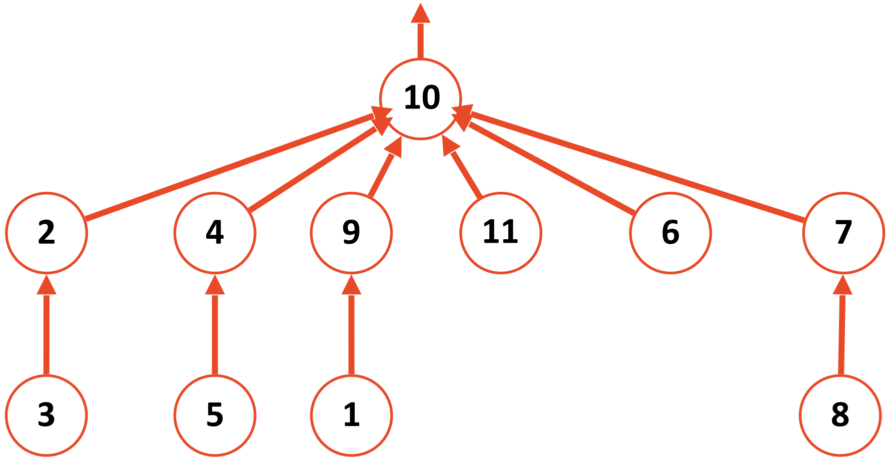
Minimizing number of O(1) operations

O(1) kinda sorta

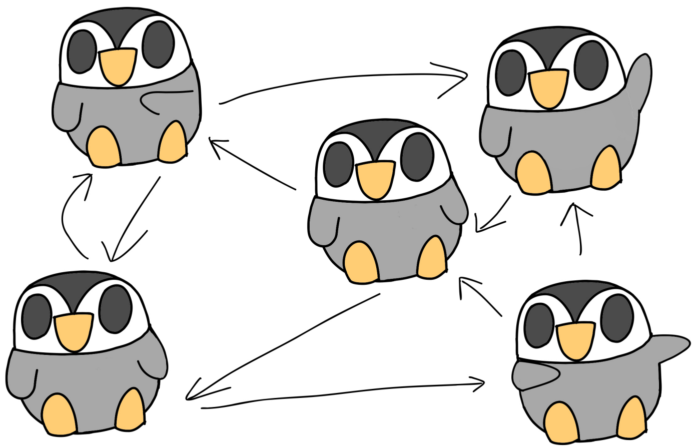
```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else {
    int root = find( s[i] );
    s[i] = root;
    return root;
  }
}
```

taking advantage of arrays



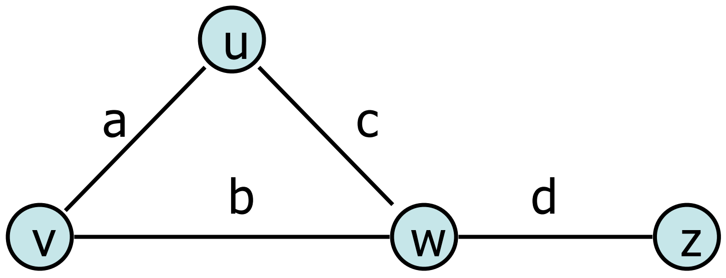


Graphs



Graph Implementation: Edge List $|V| = n, |E| = m$

The equivalent of an 'unordered' data structure



u
v
w
z

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

Vertex Storage:

An optional list of vertices

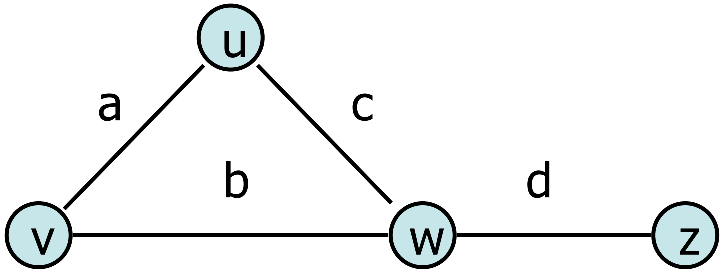
Edge Storage:

A list storing edges as (V1, V2, Weight)

Most graphs are stored as just an edge list!

Graph Implementation: Adjacency Matrix

|V|= n, |E|= m



Vertex Storage:

A hash table of vertices

Implicitly or explicitly store index

u	0
v	1
w	2
z	3

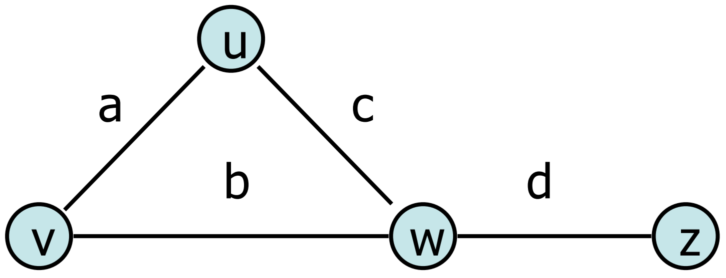
	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

Edge Storage:

A |V| x |V| matrix of edges

Weight is stored at position (u, v)

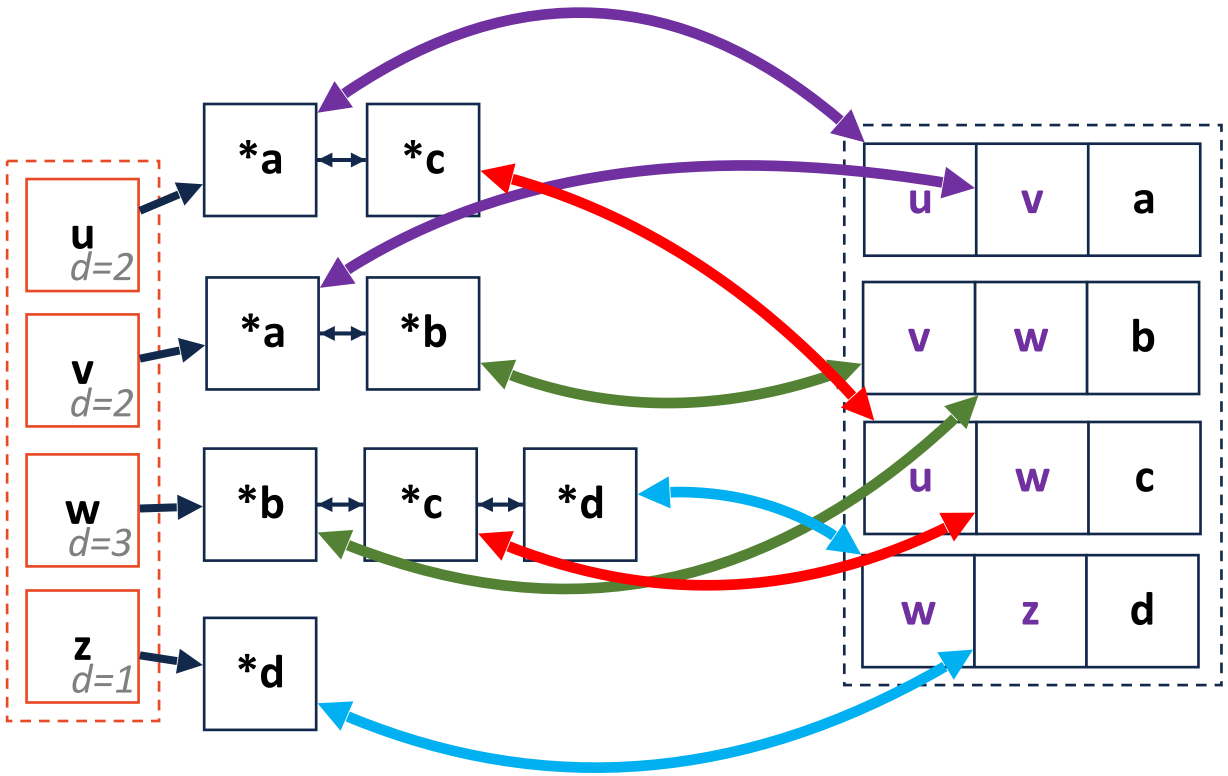
Adjacency List



Vertex Storage:

A bidirectional linked list with size variable

Each node is a pointer to edge in edge list



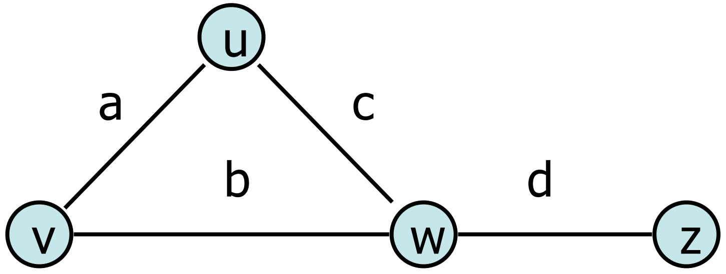
Edge Storage:

A list of (v1, v2, weight) edges

Also store pointers back to nodes

Adjacency List

|V|= n, |E|= m

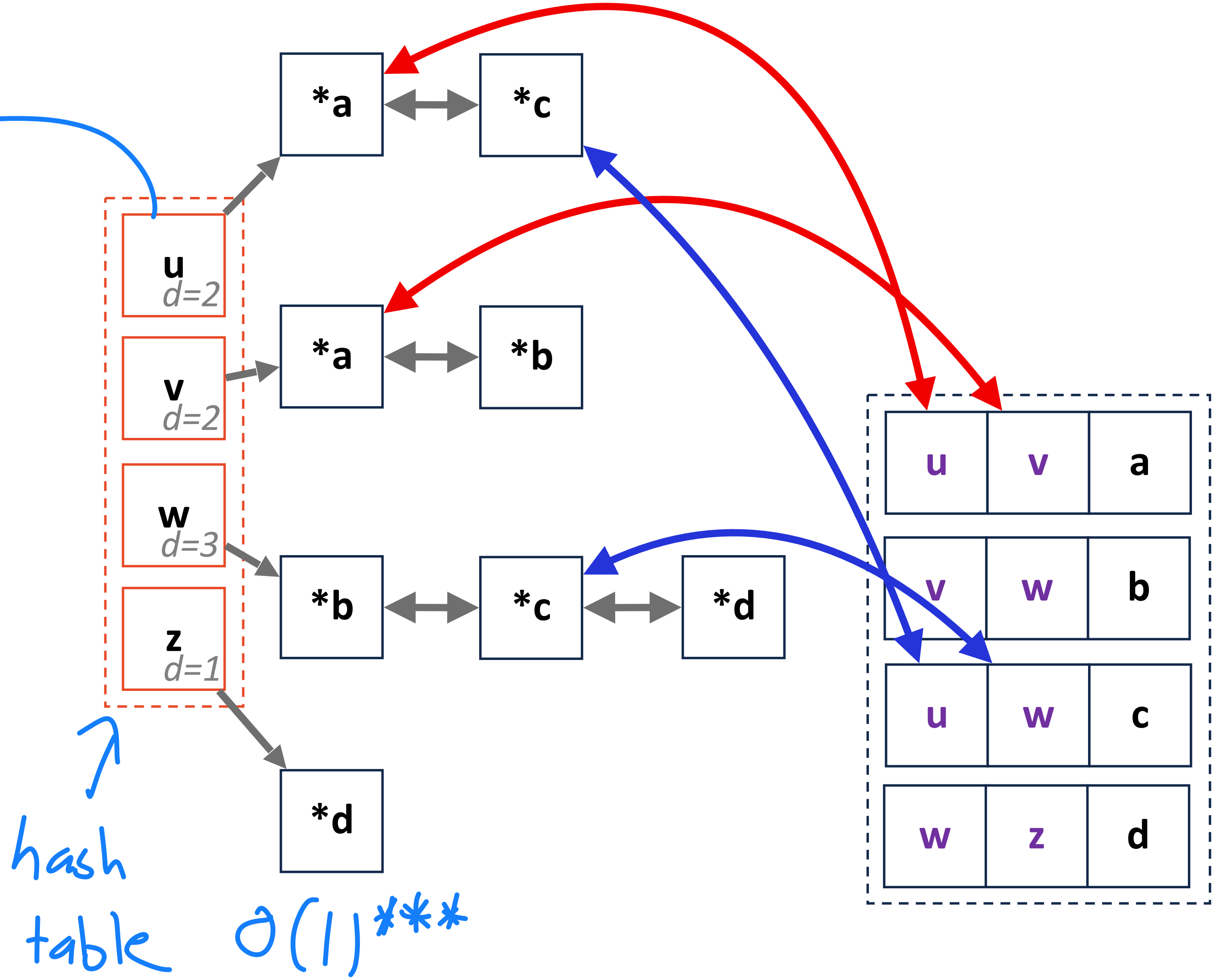


Adj List Node:

*Prev	*Edge	*Next

Edge List:

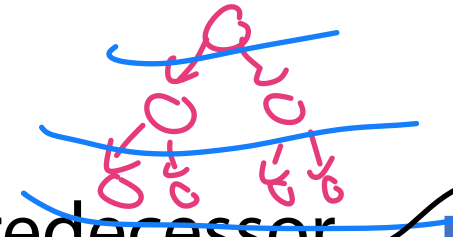
V1	V2	Weight
*V1	*V2	



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

Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+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))

Traversal: BFS



Graph implementation stores table

Vertex Node has member variable (depth, pred

Initialize queue / depth / predecessor

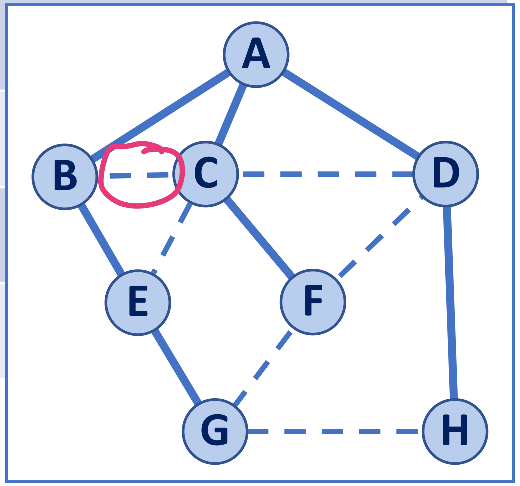
or array!

or map!

While queue not empty:

- Remove front vertex of queue
- Check if edge connects to new vertex
  - Set dist / pred if new vertex
- Add unvisited edges to queue

v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	1	A	A B D E F
D	1	A	A C F H
E	2	B	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G



~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~ ~~H~~ ~~G~~

Cross edges have meaning

↳ we already saw that vertex through a shorter path

↳ Dist between vertices linked by cross is ≤ 1

Traversal: BFS

$O(n+m)$ $|V| = n$ $|E| = m$

Initialize queue / depth / predecessor

While queue not empty:

$\boxed{O(mn)}$

Remove front vertex of queue

Check if edge connects to new vertex

Set dist / pred if new vertex $+O(n)$

Add unvisited edges to queue

Given vertex, get all edges

$O(m)$ edgs

$O(n)$ matrix

$O(\deg v)$ adj

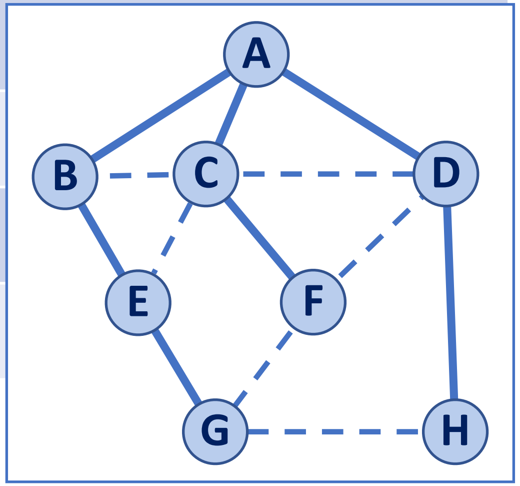
n ✗

$$\sum_v \deg(v) = 2|E| = m = O(m)$$

sparse   dense

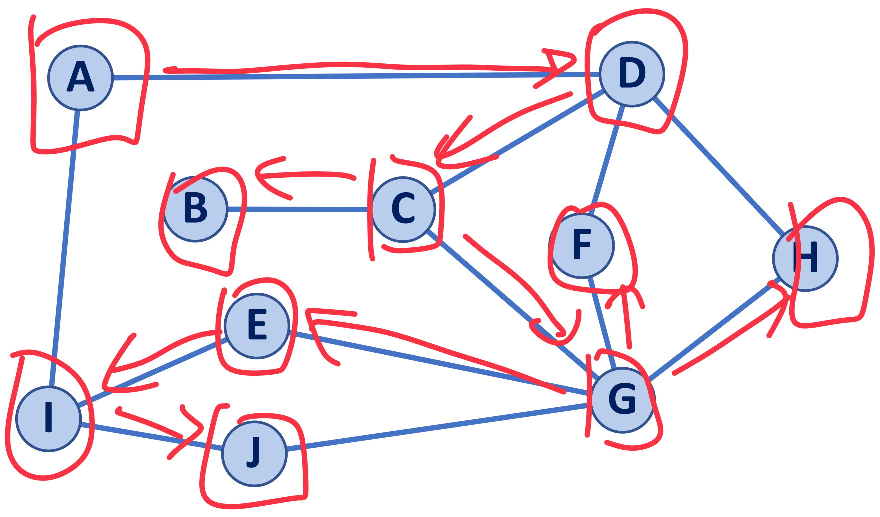
$$n-1 \le m \le n^2$$

v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	1	A	A B D E F
D	1	A	A C F H
E	2	B	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G



~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~ ~~H~~ ~~G~~

Traversal: DFS



0) Initialize dist/ pred

1) Init stack
↳ init w/ root

2) While stack not empty
↳ Peek A & get 1 unvisited child
↳ Add child to stack
↳ If no children unvisited
Pop from stack

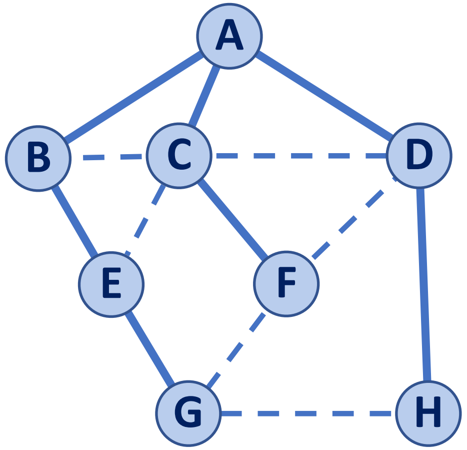
~~H~~
~~E~~
~~J~~
~~I~~
~~F~~
G
~~B~~
C
D
Bottom A

Efficiency: DFS vs BFS

(Traversal)

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

BFS: $O(n+m)$



V vertices

A B C D E F H G

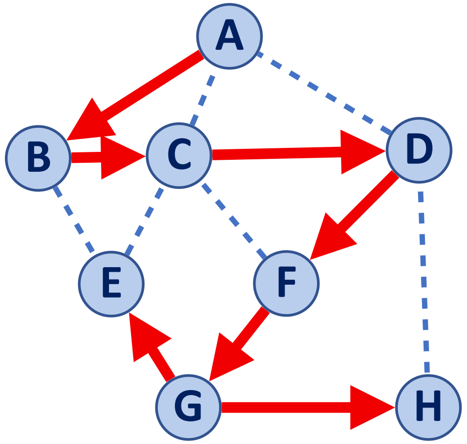
All neighbors

↳ width of graph

each deg(v)

$\sum_v deg(v) = 2|E|$

DFS: $O(n+m)$



V vertices

A B C D F G E H

↳ Longest path

each deg(v)

Summary: DFS and BFS

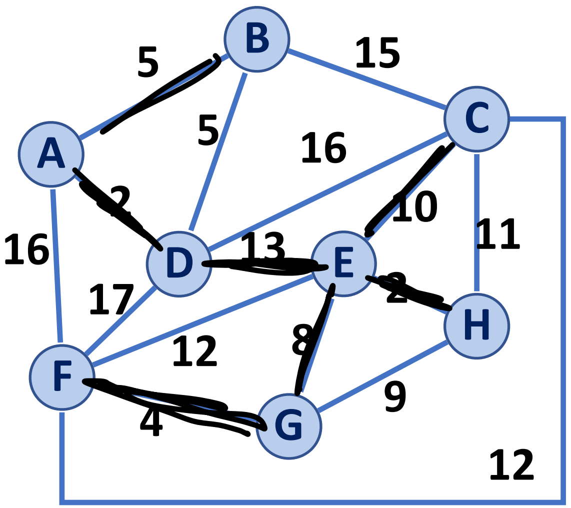
$|V| = n, |E| = m$

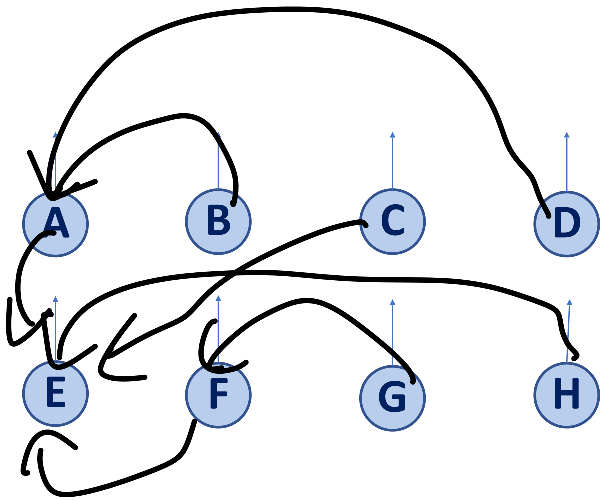
Both are **O(n+m)** traversals! They label every edge and every node

BFS	**DFS**
Solves unweighted MST	Solves unweighted MST
Solves shortest path	
Solves cycle detection	Solves cycle detection
Memory bounded by width	Memory bounded by longest path
	↳ considered better in memory

Kruskal's Algorithm

(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) Build a **priority queue** on edges

↳ min heap

↳ sorted list

2) Build a **disjoint set** on vertices

↳ All vertices start as own set

3) Repeat take min edge

↳ If connect two sets

↳ union sets

↳ record edge

4) Stop when:

– n-1 nodes recorded

– I have one disjoint set

Kruskal's Algorithm

```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

1) Build a **priority queue** on edges

2) Build a **disjoint set** on vertices

3) Repeatedly find min edge

    If edge connects two sets

    Union and record edge

4) Stop after n-1 edges recorded

Kruskal's Algorithm

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

Priority Queue:	Heap	Sorted Array
Building :7	$O(m)$	$O(m \log m)$
Each removeMin :12 $m \times$	$O(\log m)$	$O(1)$

```
1   KruskalMST(G):
2     DisjointSets forest                              O(n)
3     foreach (Vertex v : G.vertices()):
4       forest.makeSet(v)
5
6     PriorityQueue Q     // min edge weight
7     Q.buildFromGraph(G.edges())
8
9     Graph T = (V, {})
10                                               mx
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) )     O(1)
17
18    return T
19
```

$m + m\log m$ vs $m\log m + m$

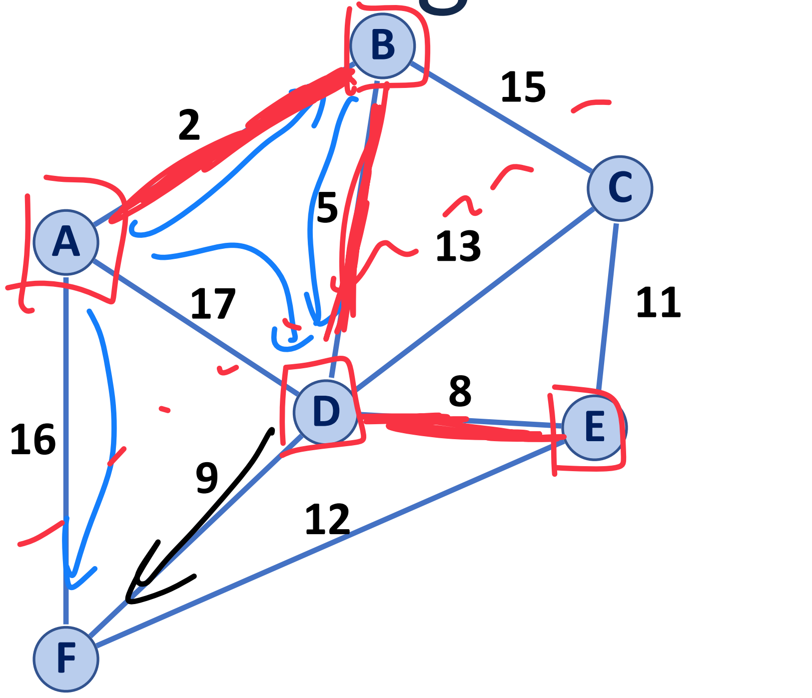
why heap good?

↳ what if edge weight changes?

why sorted array good?

↳ Sorted array <u>not</u> destroyed when used = if we could use array later, this is better!

Prim's Algorithm



11,E

A	B	C	D	E	F
0	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~

2,A ~~15,C~~ ~~13,D~~ ~~17,A~~ 5,B 8,D ~~16,A~~ 9,D

```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

Init

update all neighbors if new smaller edge

Prim's Algorithm

Sparse Graph: $\boxed{n \sim m}$

↳ heap is better

Dense Graph: $\boxed{m \sim n^2}$

↳ unsorted array better

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

This is updating

$n-1 \le m \le n^2$

$m = n^2$

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

MST Algorithm Runtime:

Kruskal's Algorithm:
$O(n + m \log(n))$

Prim's Algorithm:
$O(n \log(n) + m \log(n))$

Sparse Graph: $m \sim n$

Dense Graph: $m \sim n^2$

Dijkstra's Algorithm (SSSP)

Assume heap / Fib

What is the running time of Dijkstra's Algorithm?

↳ This is Prim!

$O(n + \underbrace{\boxed{n \log n} + \boxed{m}}_{\text{Dom term}})$

Find min

update

@15 + @18: $\sum_{v} \deg(v) = 2m$

Total # edge updates is m

```
    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)

nx

O(log n)

O(1*)

Dijkstra's Algorithm (SSSP)

Dijkstras Algorithm works only on non-negative weights

Optimal implementation:

Fibonacci Heap

If dense, unsorted list ties

Optimal runtime:

Sparse: O(m + n log n)

Dense: O(n²)

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m

  return T
```

(Basically Prim)

← This changes

Floyd-Warshall Algorithm

Running time? $O(n^3)$ operation!

↳ Easy to code (multithreadable!)

↳ Handles neg weight (but not cycle)

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex u : G):           n×
    foreach (Vertex v : G):         n×
      foreach (Vertex w : G):       n×
        if d[u, v] > d[u, w] + d[w, v]:
          d[u, v] = d[u, w] + d[w, v]      O(1)
```

matrix

Final thoughts on Graphs

Graphs have a large space of **possible coding questions**

You should be able to solve common graph questions

- Make sure you can use graphs to find all neighbors
- Make sure you can use graphs to solve path questions

Consider how these fundamental skills can be challenged

- What if I had labels on nodes and I need to find specific ones?
- What if I need to label nodes or edges with specific properties?
- Can I handle weights? Directions?

Probability in CS

Fundamentals of Probability

Imagine you roll a pair of six-sided dice. What is the expected value?

A **random variable** is a function from events to numeric values.

D1 is value of first dice →

DBoth is value of D1 + D2

The **expectation** of a (discrete) random variable is:

$$E[D1] = \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \ldots \approx 3.5$$

$$E[X] = \sum_{x \in \Omega} Pr\{X = x\} \cdot x$$

$$E[DBoth] = \frac{1}{36} \cdot 2 + \frac{1}{36} \cdot (1+2) + \ldots \approx 7$$

Next Class: Randomized Data Structures

Sometimes a data structure can be **too ordered / too structured**

Randomized data structures rely on **expected** performance

Randomized data structures 'cheat' tradeoffs!

↳ Add inaccuracy for speed gains

Probabilistic Data Structures

Randomized Algorithms

A **randomized algorithm** is one which uses a source of randomness somewhere in its implementation.

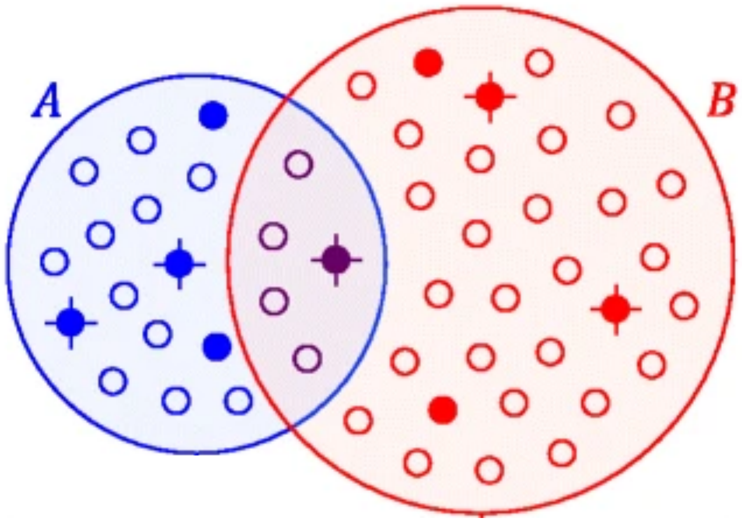
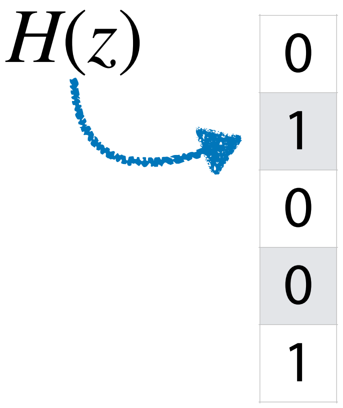
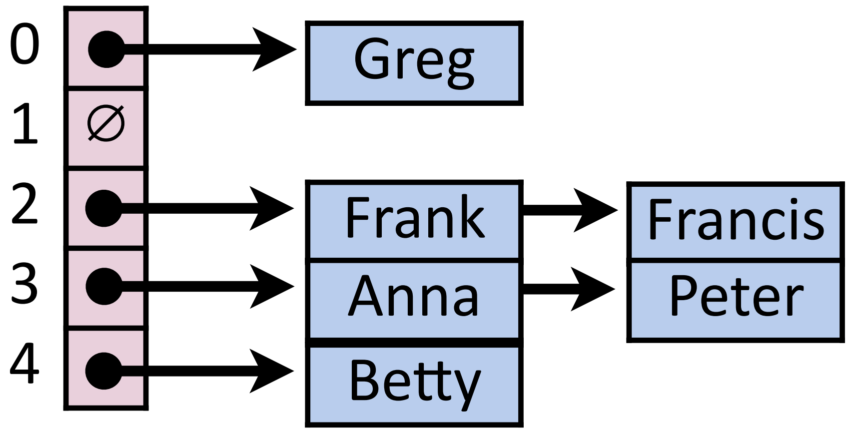


Figure from Ondov et al 2016





$H(x)$	0	2	1	0	0	4	0	2	0	6
$H(y)$	1	0	2	3	1	0	3	4	0	1
$H(z)$	2	1	0	2	0	1	0	0	7	2

A Hash Table based Dictionary

User Code (is a map):

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

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

1. A hash function — Assigns numeric (positive int) address to any key

   Key -> Hash Value (Address)

2. A data storage structure — Array — very good at lookup given **index**

   Hash Value (Address) is an index!

3. A method of addressing *hash collisions*

Two different keys, same hash value

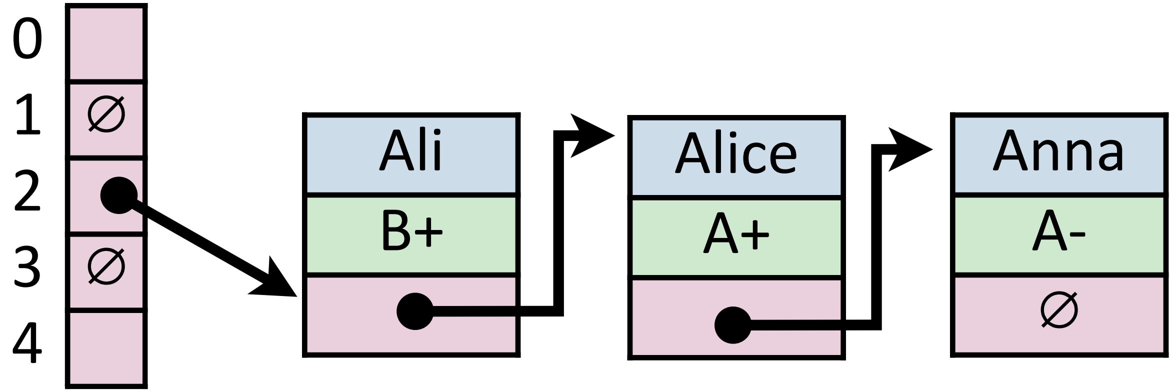
Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

- **Open Hashing:** store *k,v* pairs externally

Such as a linked list

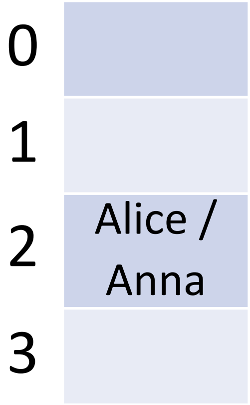
Resolve collisions by adding to list



- **Closed Hashing:** store *k,v* pairs in the hash table

Everything stored in one list

How to store collisions? Unclear!



Simple Uniform Hashing Assumption

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 to any position

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

Independent: All key's hash values are independent of other keys

Separate Chaining Under SUHA

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

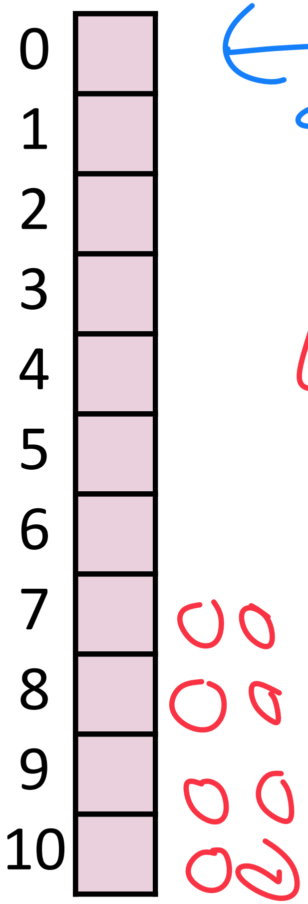
Find runs in: O(1+α).

$\alpha = \frac{n}{m}$

a constant we control

Insert runs in: O(1).

Remove runs in: O(1+α).



expected length

at every position

Load Factor (α)

n/m

Running Times (Expectation under SUHA)

Open Hashing: $0 \leq \alpha \leq \infty$ (Length of chain)

insert: $\underline{\quad 1 \quad}$.

find/ remove: $\underline{\quad 1 + \alpha \quad}$.

Closed Hashing: $0 \leq \alpha < 1$ (fraction full)

insert: $\underline{\quad \frac{1}{1-\alpha} \quad}$.

find/ remove: $\underline{\quad \frac{1}{1-\alpha} \quad}$.

Observe:

* **- As α increases:**

OH: $\alpha \to \infty$, runtime $\to \infty$

CH: $\alpha \to 1$, runtime $\to \infty$

* **- If α is constant:**

OH is constant

CH is constant

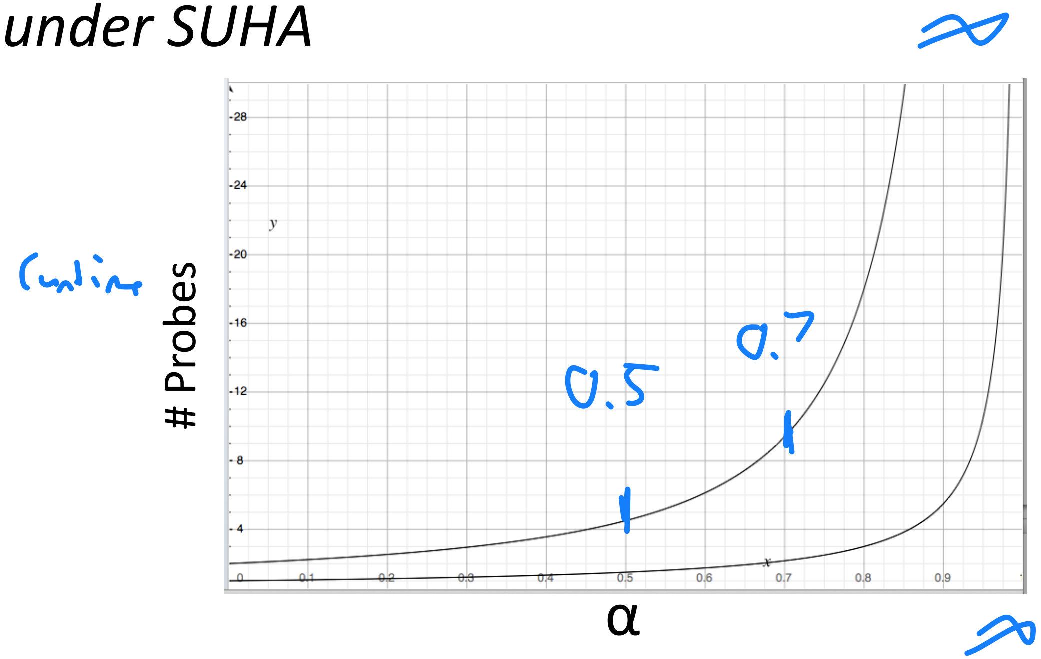
$O(1)^*$:)

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

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

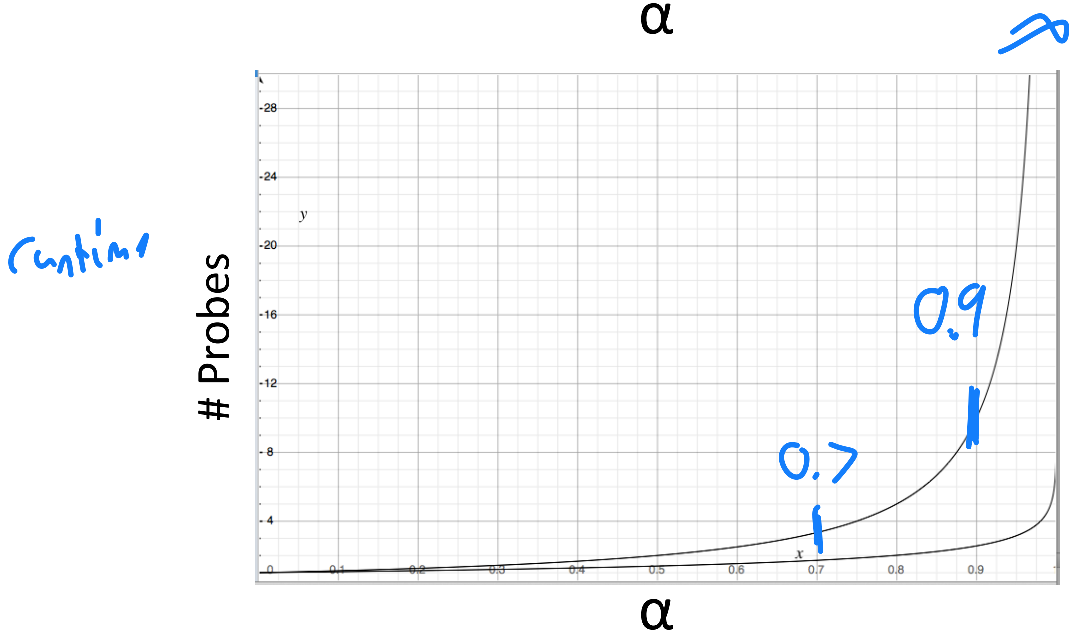
Linear Probing:

- 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)**



When do we resize?

Linear ~0.7-0.8

Double ~0.7-0.9

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	$O(n)$	$O(n)$	$O(n)$

Bloom Filter

A probabilistic data structure storing a set of values

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

0
0
1
0
0
1
0
1
0
0

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

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

Delete is not possible (yet)!

Probabilistic Accuracy in a Bloom Filter

	Bit Value = 1	**Bit Value = 0**
Item Inserted	$H(z)$ → 0, 1, 0, 0, 1 **'Yes'** True Positive	$H(z)$ → 0, 0, 0, 0, 1 Inserted z **'No'** ~~False Negative~~
Item NOT inserted	0, 1, 0, 0, 1 **'Yes'** Hash collisions False Positive	0, 0, 0, 0, 1 **'No'** True Negative

Not possible!

↳ B/c no removal

Bloom Filter: Error Rate

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

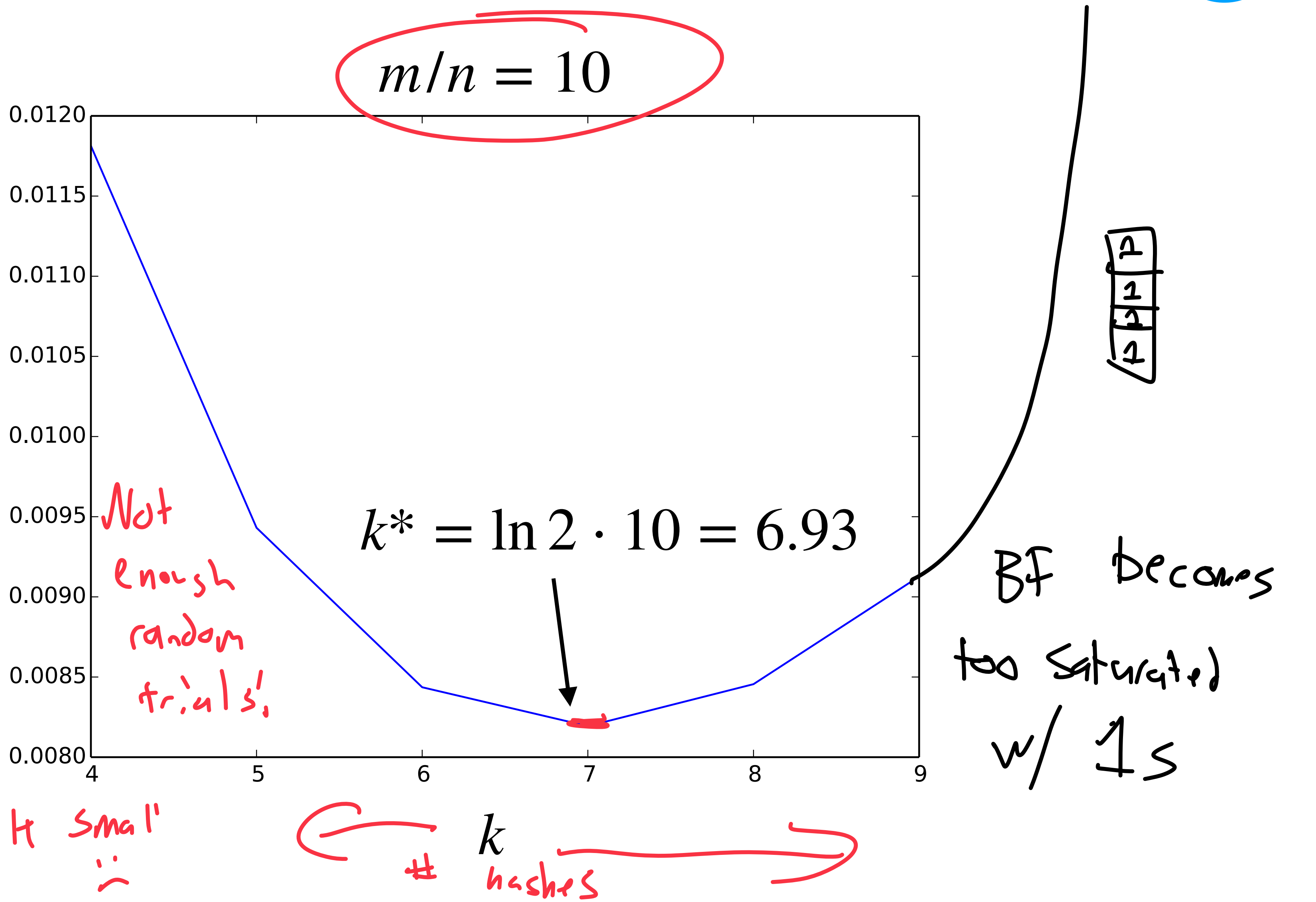


Figure by Ben Langmead

Cardinality Estimation

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

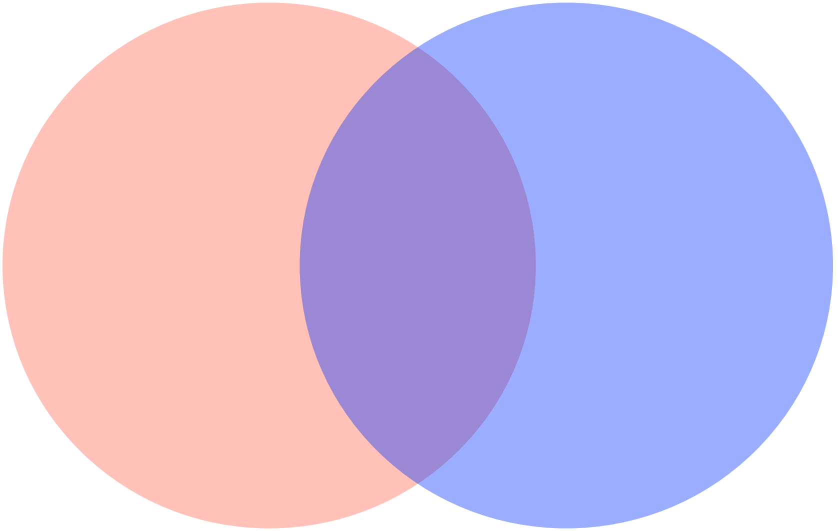
Conceptually: If we scatter N points randomly across the interval, we end up with $N + 1$ partitions, each about $1000/(N+1)$ long

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!

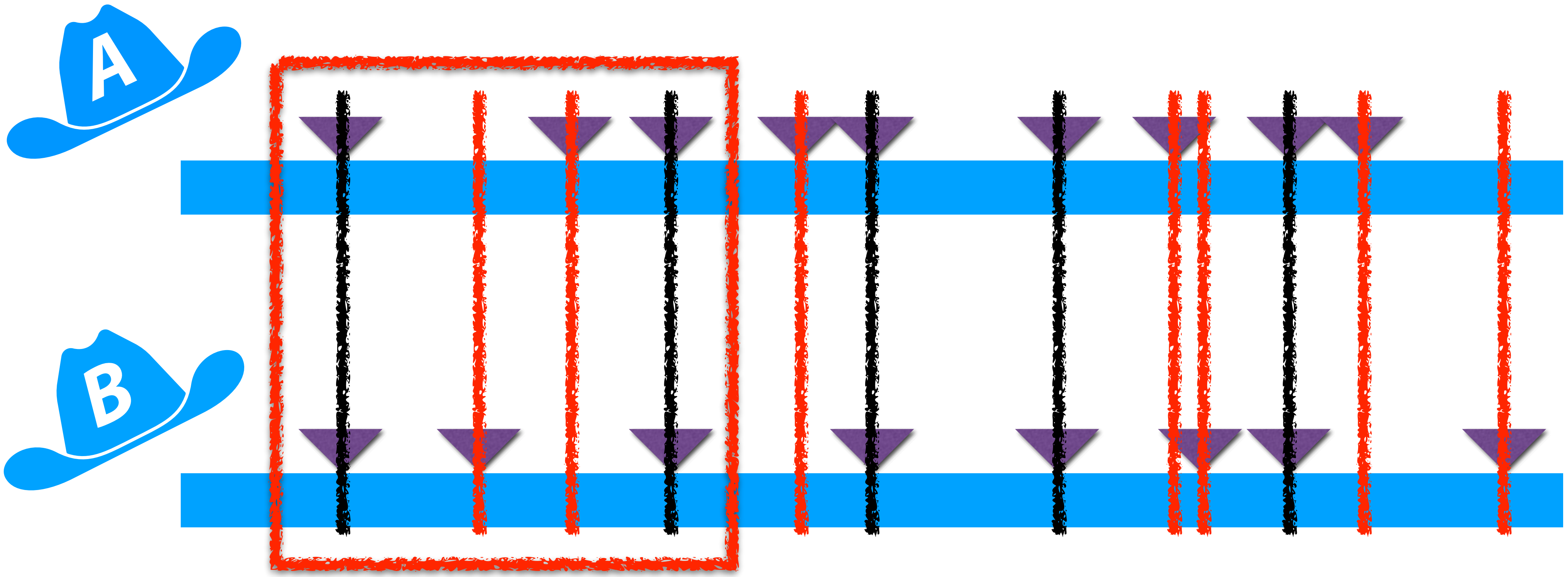
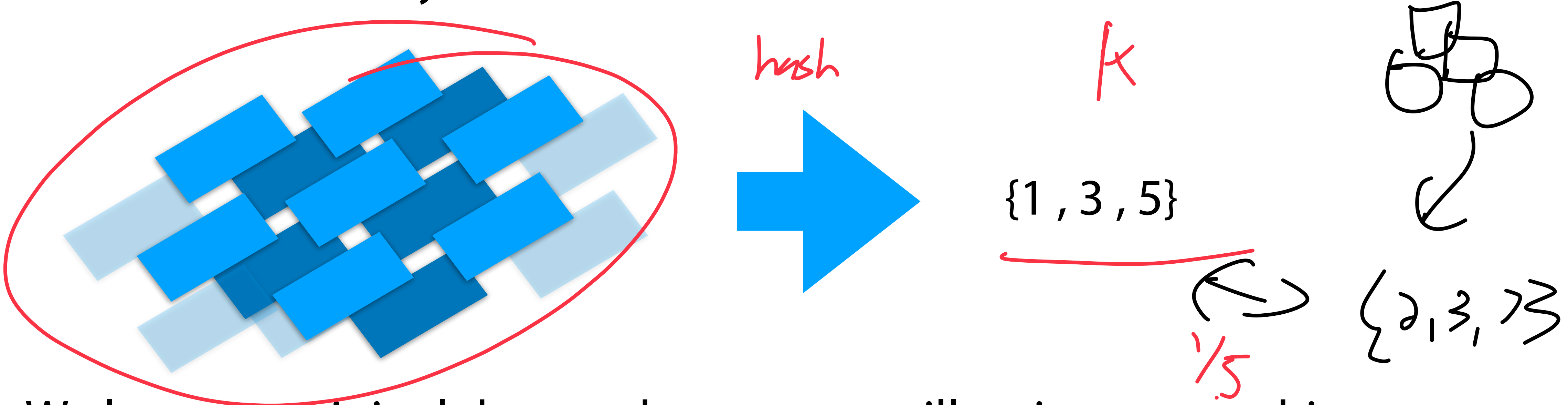


Image inspired by: Ondov B, Starrett G, Sappington A, Kostic A, Koren S, Buck CB, Phillippy AM. **Mash Screen: high-throughput sequence containment estimation for genome discovery**. *Genome Biol* 20, 232 (2019)

MinHash Sketch

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



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

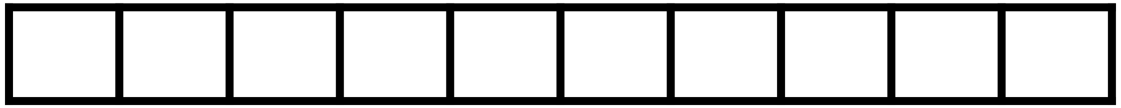
1. Cardinality (# of items)

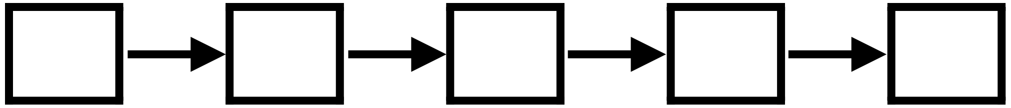
2. Set Similarity

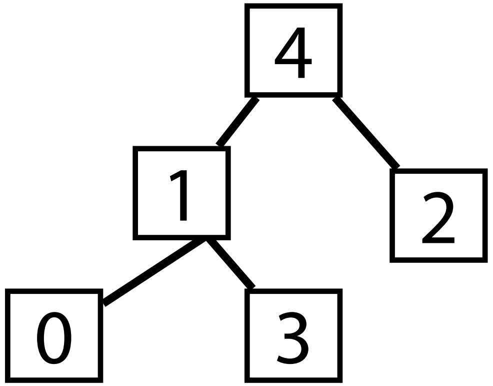
Questions?

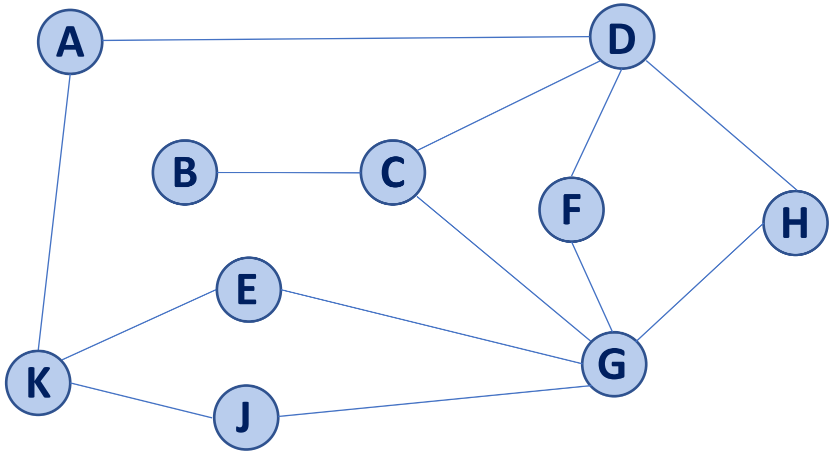
CS 225 — Course Goals

Understand foundational data structures and algorithms







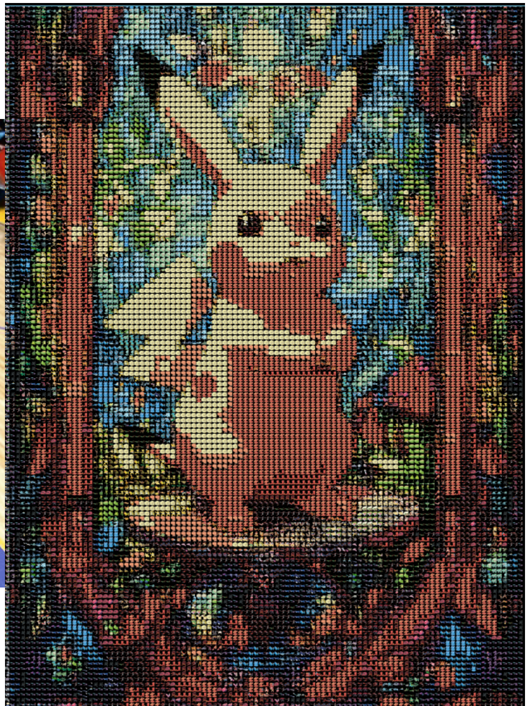


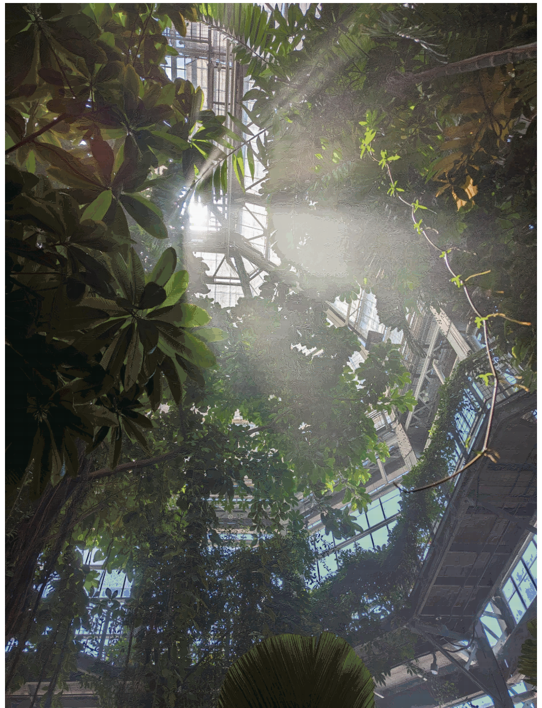
CS 225 — Course Goals

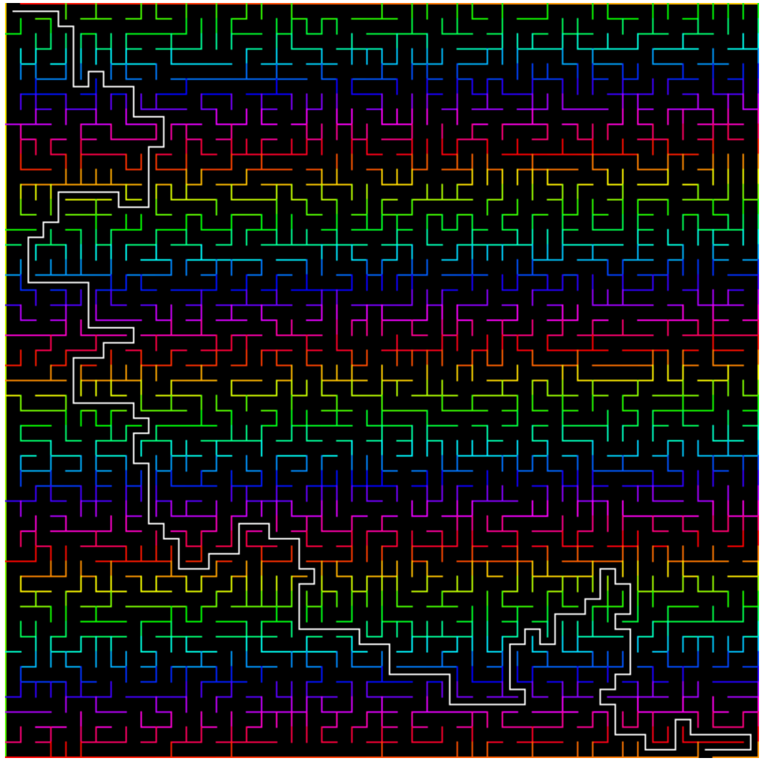
Justify appropriate algorithms for complex problems

Decompose problem into supporting data structures

Analyze efficiency of implementation choices

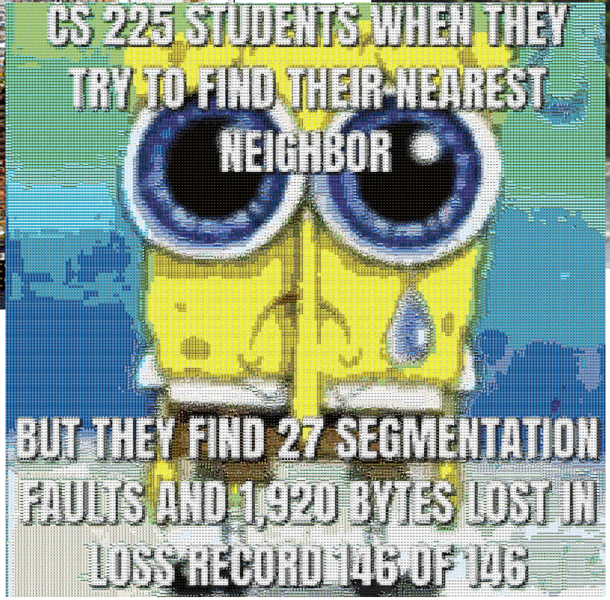






CS 225 — Course Goals

Implement intermediate difficulty problems in C++



CS 225 — Course Goals

Understand foundational data structures and algorithms

Justify appropriate algorithms for complex problems

Implement intermediate difficulty problems in C++

Improve your foundation of CS theory

Good luck on your finals!