

Data Structures Review

CS 225
Brad Solomon

December 8, 2025



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

Announcements

Fill out FLEX Evaluation!

Interested in being a CA? Apply now!

<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.

Brad's suggested review strategies:

- 1) Go through lecture content (focus on review slides)

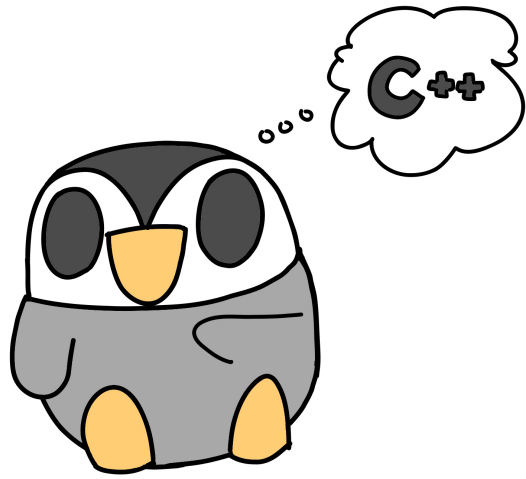
If there's material you don't remember fully, rewatch lecture

- 2) Go through practice exams once

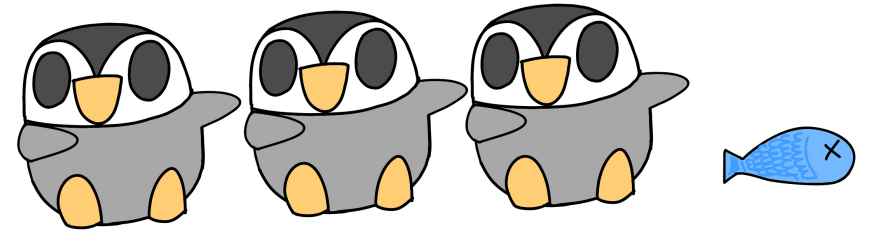
If there's material you are struggling with, focus efforts here

- 3) Review course assignments

Don't look at your solution until after attempting it from scratch



Lists



List Implementation

September 10 (Quack Lecture)



	Singly Linked List	Array
Look up <u>arbitrary</u> location ↳ random access	$O(n)$	$O(1)$ ☺
Insert after given element	$O(1)$ ☺	$O(n)$
Remove after given element	$O(1)$ ☺	$O(n)$
Insert at arbitrary location	Find is $O(n)$ Mod is $O(1)$ $O(n)$	Find is $O(1)$ Mod is $O(n)$ $O(n)$
Remove at arbitrary location	$O(n)$	$O(n)$
Search for an input value	$O(n)$	$O(n)$

Special Cases:

insert Front
remove

(head)

insert Back
remove
when not full

Lists

November 7 (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:

insertFront

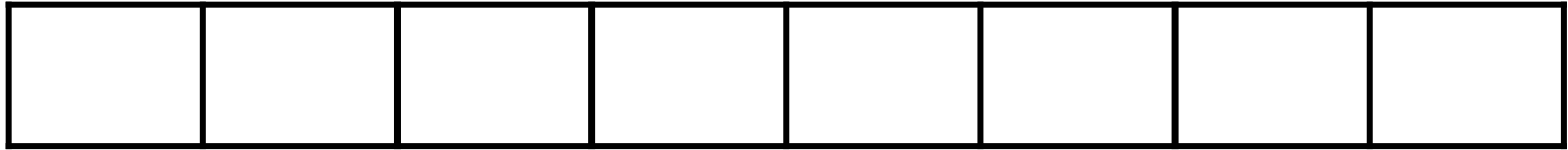
insertBack (not full)

Stack and Queue

November 7 (Review Lecture)

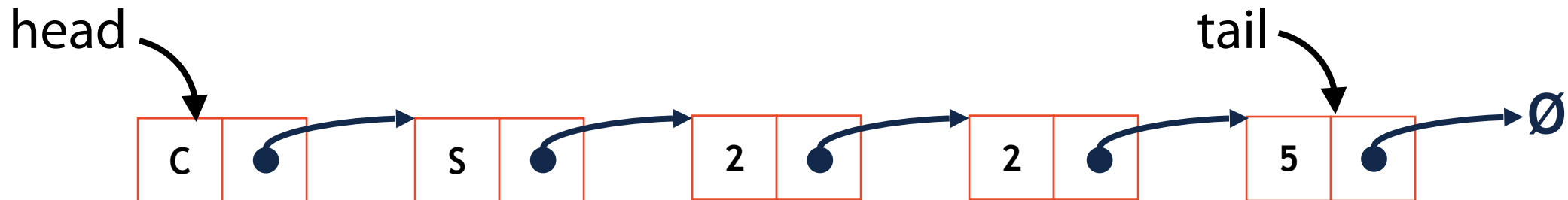
Taking advantage of special cases in lists / arrays

insertBack $O(1)$



insertFront $O(1)$

insertBack $O(1)$



Stack ADT

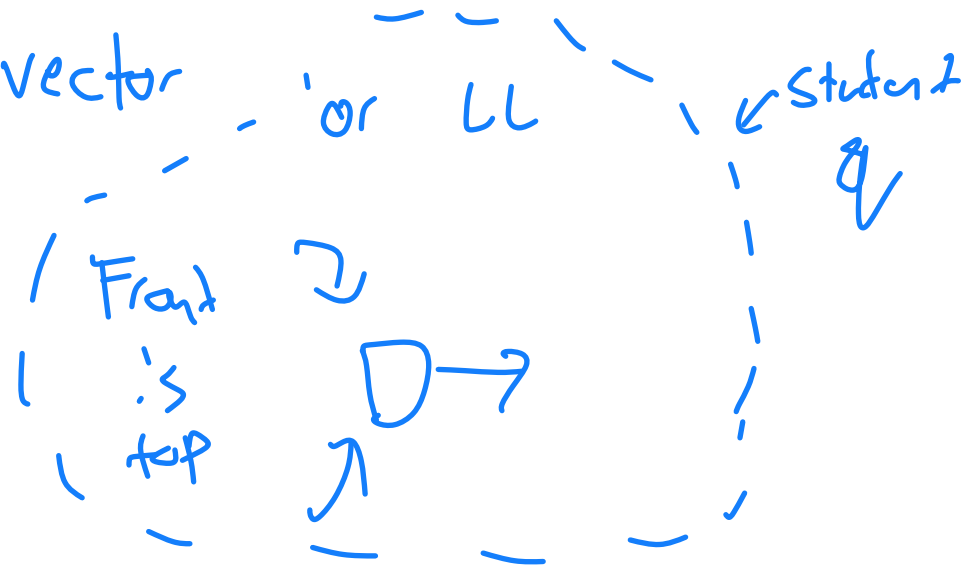
September 10 (Quacks Lecture)



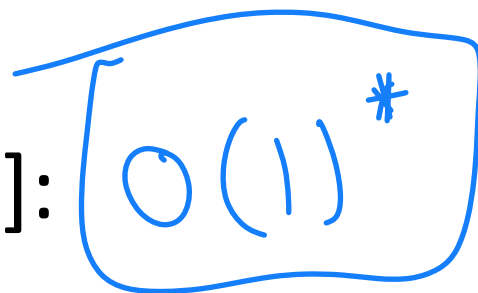
- [Order]: Last in first out (LIFO)



- [Implementation]: Trivially as vector or LL



- [Runtime]:



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

Stack ADT

- [Order]: LIFO (Last in first out)
- [Implementation]: Array (such as `std::vector`)

Linked List also works using insert / remove Front

- [Runtime]: $O(1)$ Push and Pop

If using array, $O(1)^*$ if we need to resize.



Queue ADT

- [Order]: First in first out
- [Implementation]: Trivially as LL
w/ circular queue as array
- [Runtime]: $O(1)$ * when array amortized $O(1)$

Queue ADT

- [Order]: FIFO (First In First Out)
- [Implementation]: Circular Queue as Array

Linked List also works using removeFront / insertBack or vice versa

- [Runtime]: $O(1)$

Iterators

September 12 (Queue-Iterator Lecture)

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 ++() *← get next*

const T & operator *() *← dereference*

bool operator !=(const Iterator &) *← compare iterator objects*

Iterators (225 Webpage Resources)

`it`

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

`end`

`it`

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

`end`

`it`

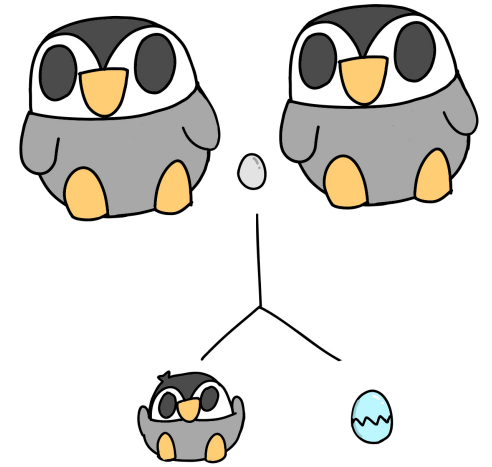
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

`end`

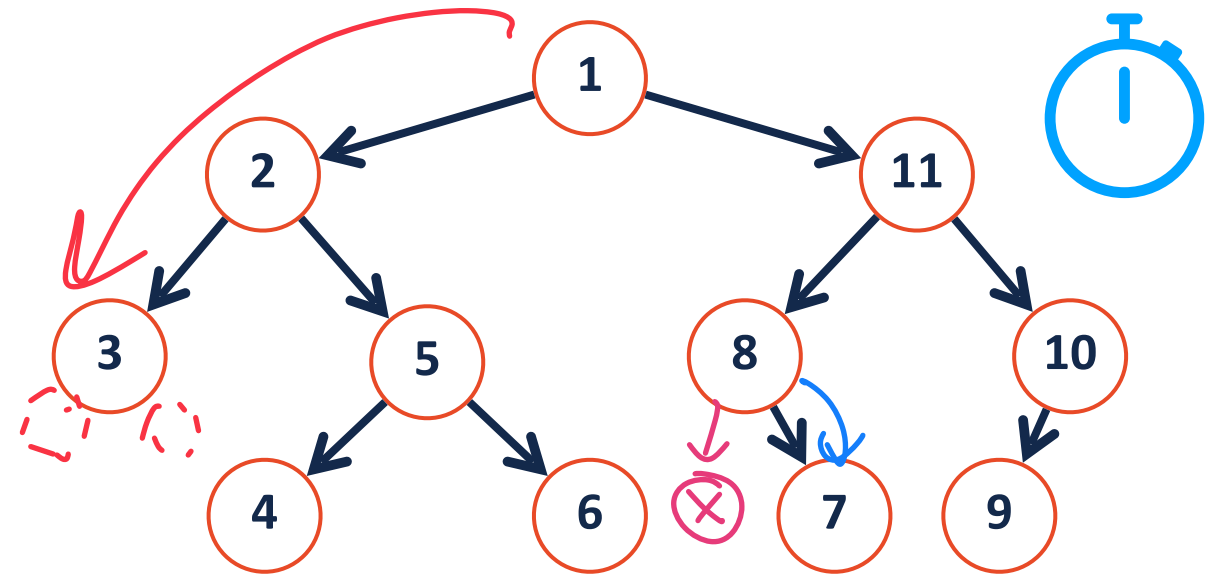
<https://courses.grainger.illinois.edu/cs225/fa2024/resources/iterators/>

Trees

Labs in this section great for reviewing fundamentals



Tree Traversals

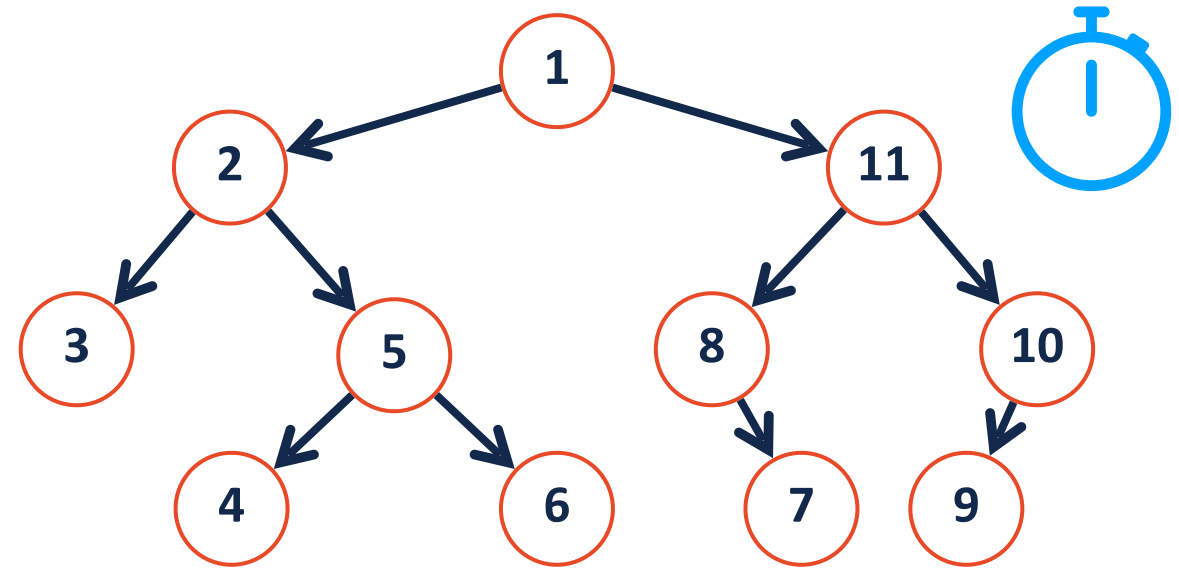


Pre-order: 1 2 3 5 4 6 11 8 7 10 9

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

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

Max size of stack $\approx$ Height of Tree

Explore as far along one path as possible before backtracking

Make a stack initialized with root

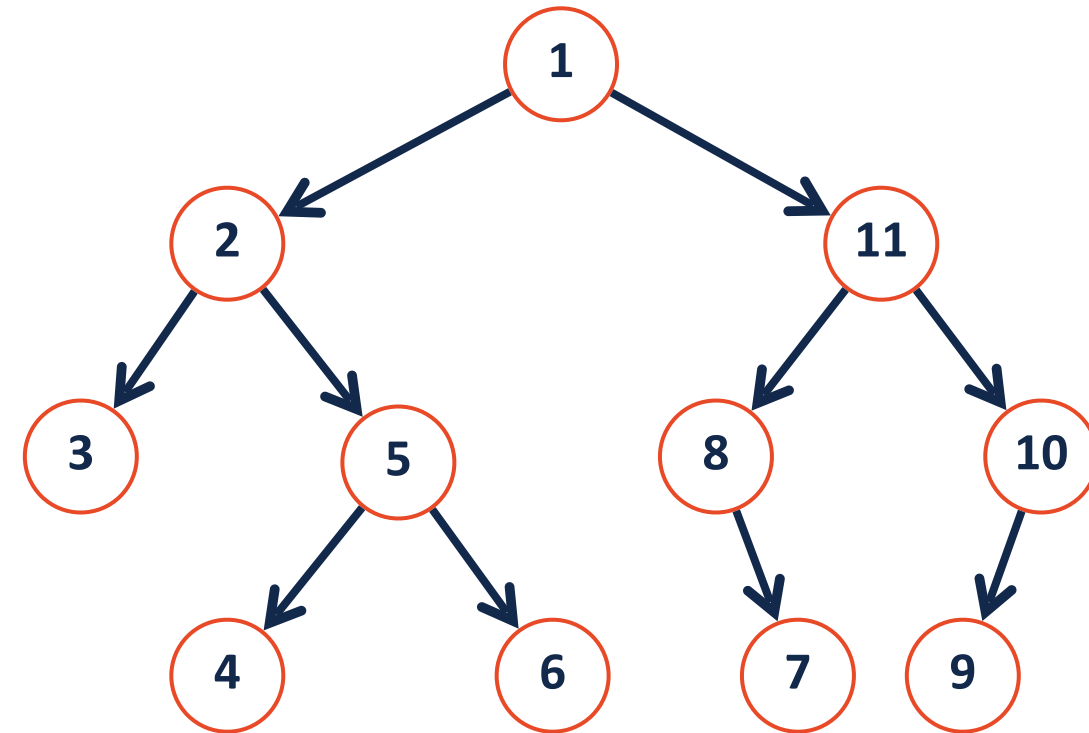
While stack isn't empty:

- Pop top element (as tmp)

- Print tmp

- Push tmp->right to stack

- Push tmp->left to stack



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

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

Breadth First Search

Max size of queue $\approx$ Width of Tree

Fully explore depth i before exploring depth $i+1$

Make a queue initialized with root

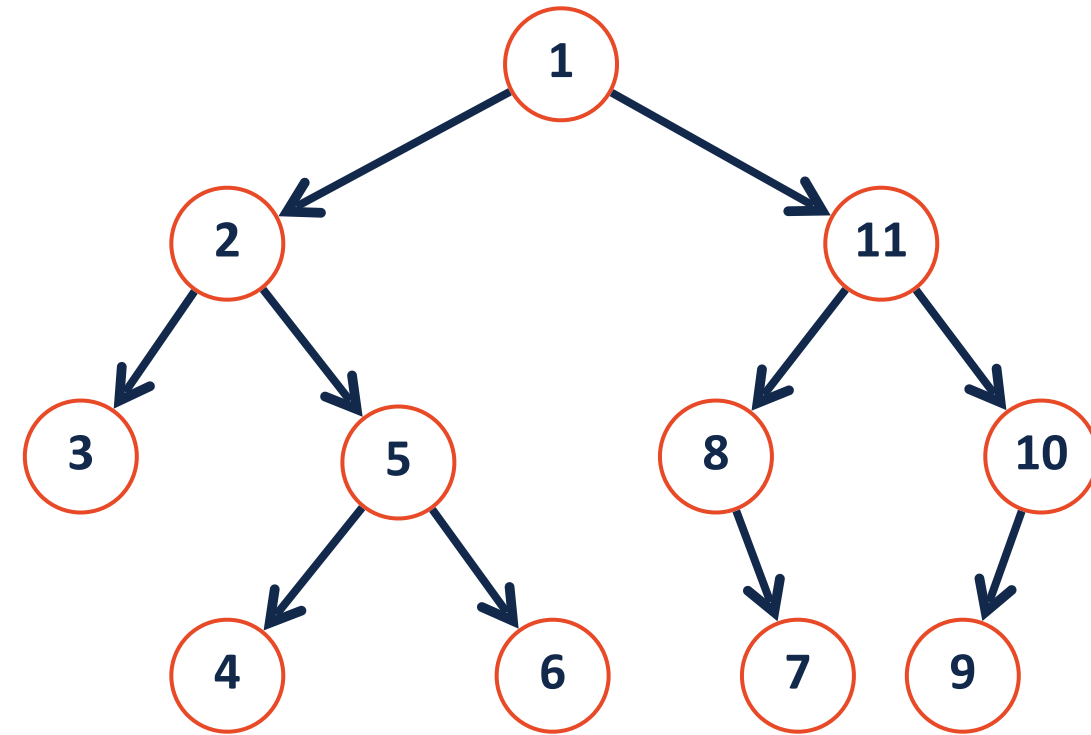
While queue isn't empty:

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)

A recursive function based around value of root:

Base Case: If root is null, return root

Let tmp = root->key()

tmp == query, return root

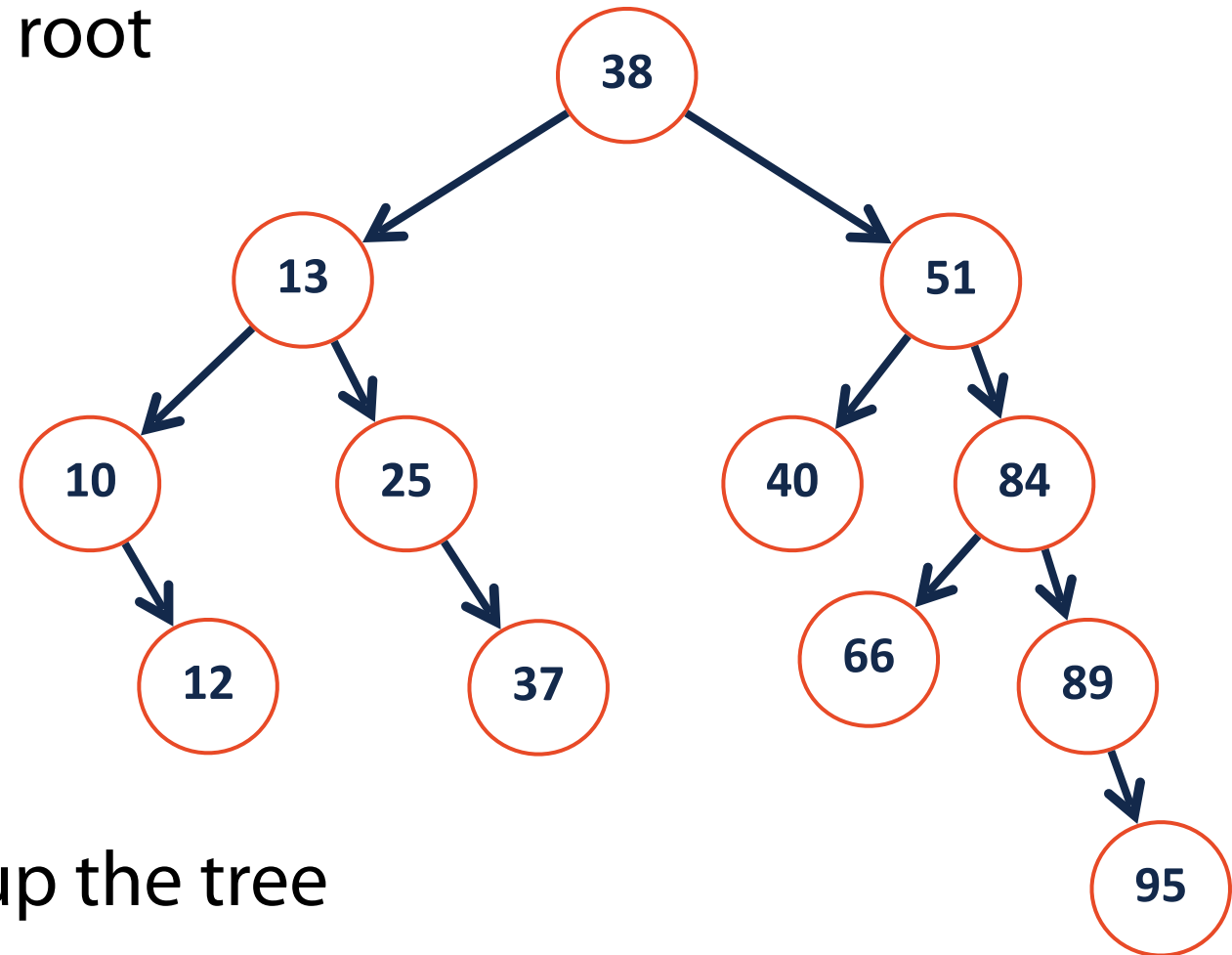
Recursion:

tmp < query, recurse right

tmp > query, recurse left

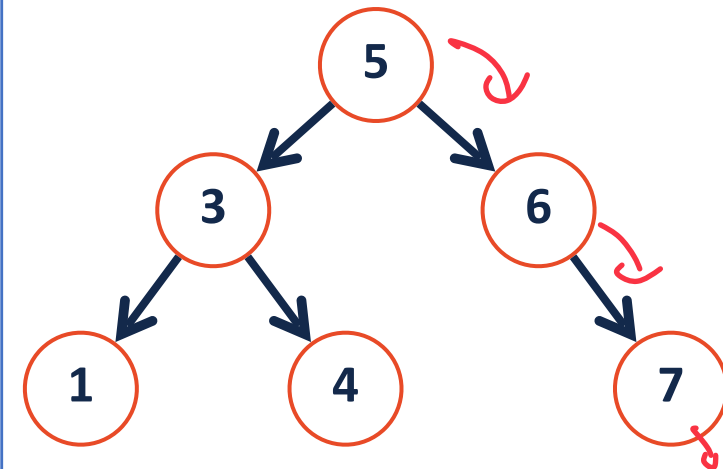
Combining:

Return the recursive value back up the tree





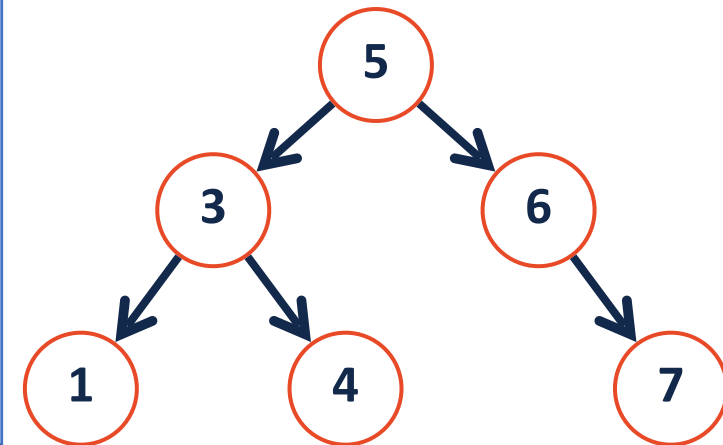
```
1 template<typename K, typename V>
2
3     TreeNode *& __find(TreeNode *& root, const K & key) {
4
5
6     // Base Case
7     if(root == nullptr || root->key == key){
8         return root;
9     }
10
11    // Recursive Step ("Combining step" is 'return')
12    if (root->key > key){
13        return __find(root->left, key);
14    }
15
16    return __find(root->right, key);
17
18
19 }
20
21
22
23
```





```
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     TreeNode *& tmp = _find(root, key); Reference to pointer!
6
7
8     tmp = new treeNode(key, val);
9
10
11
12
13 }
14
15
16
```

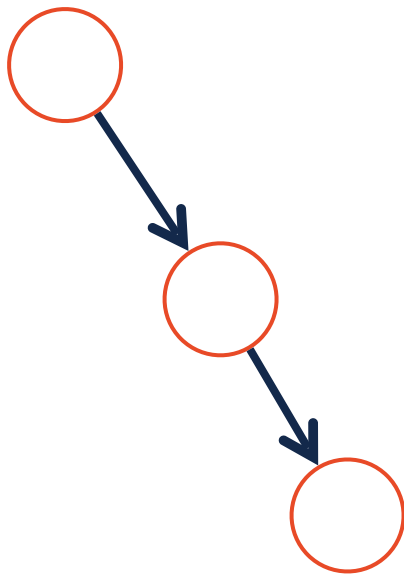


BST Analysis – Running Time

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

AVL Rotations

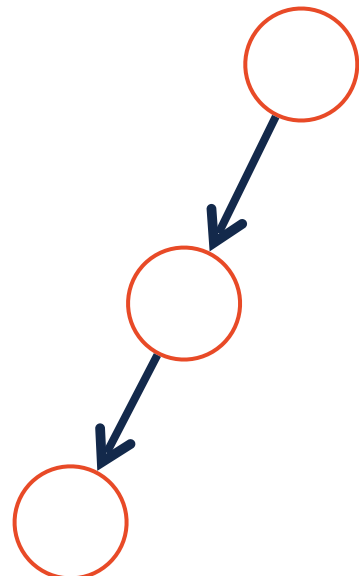
Left



Root Balance: 2

Child Balance: 1

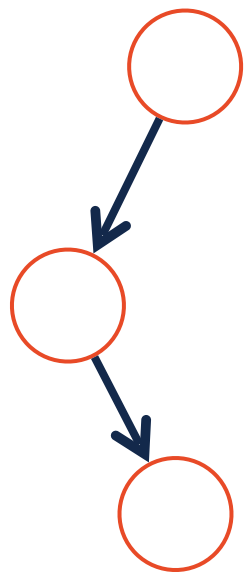
Right



-2

-1

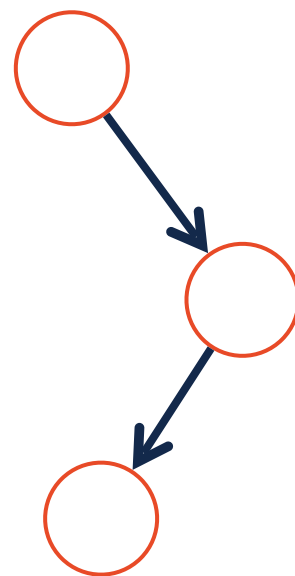
LeftRight



-2

1

RightLeft



2

-1



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: $O(h)$.

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

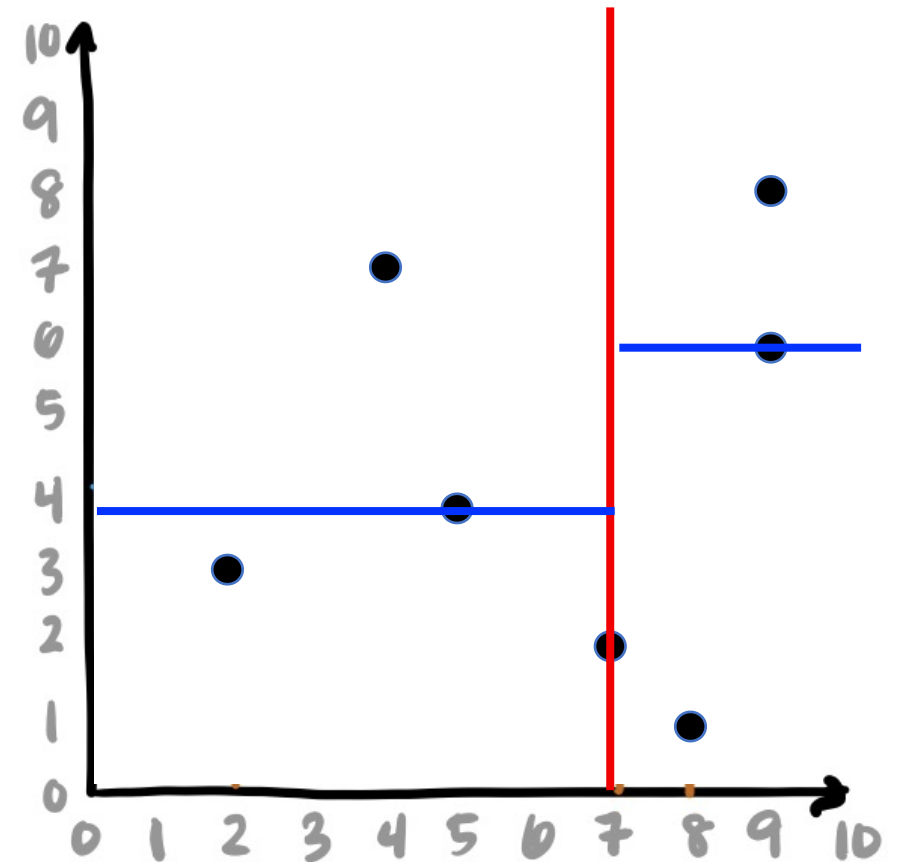
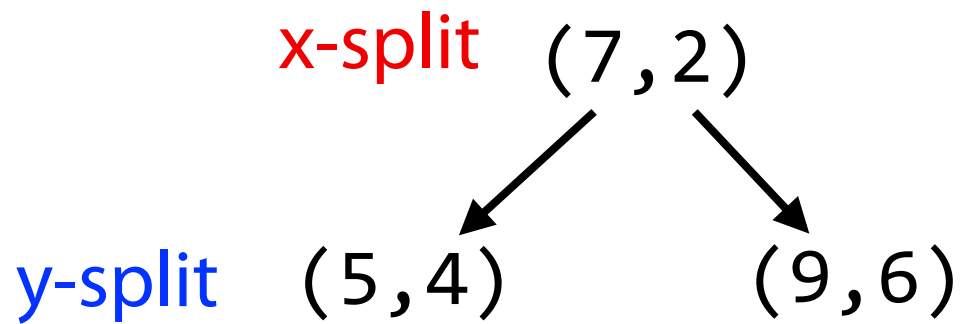
Guarantee:
1) Tree is balanced



Nearest Neighbor: k-d tree

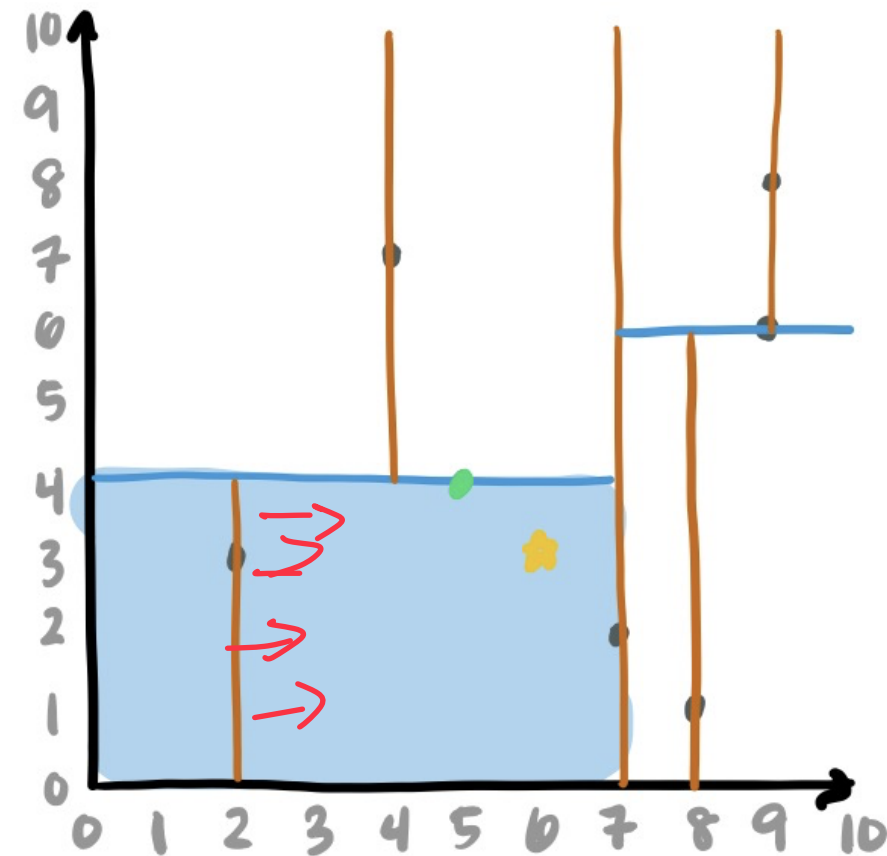
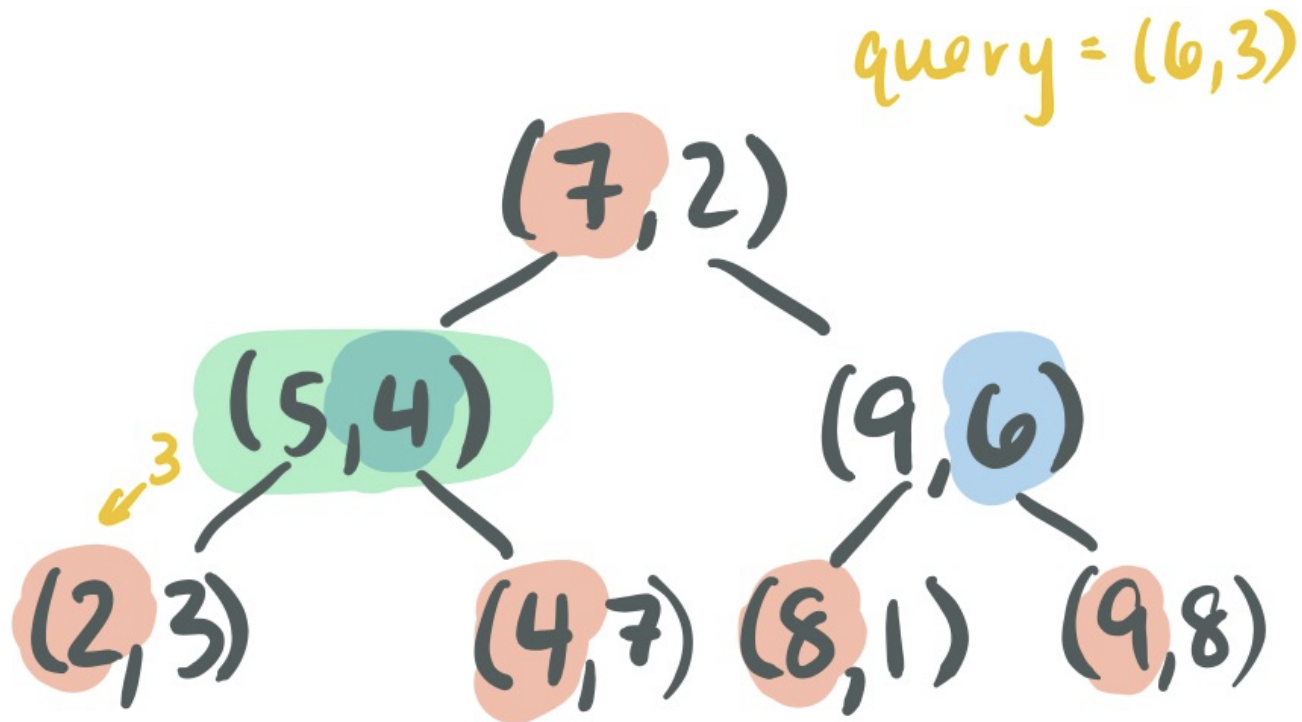
A **k-d tree** is similar but splits on points:

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



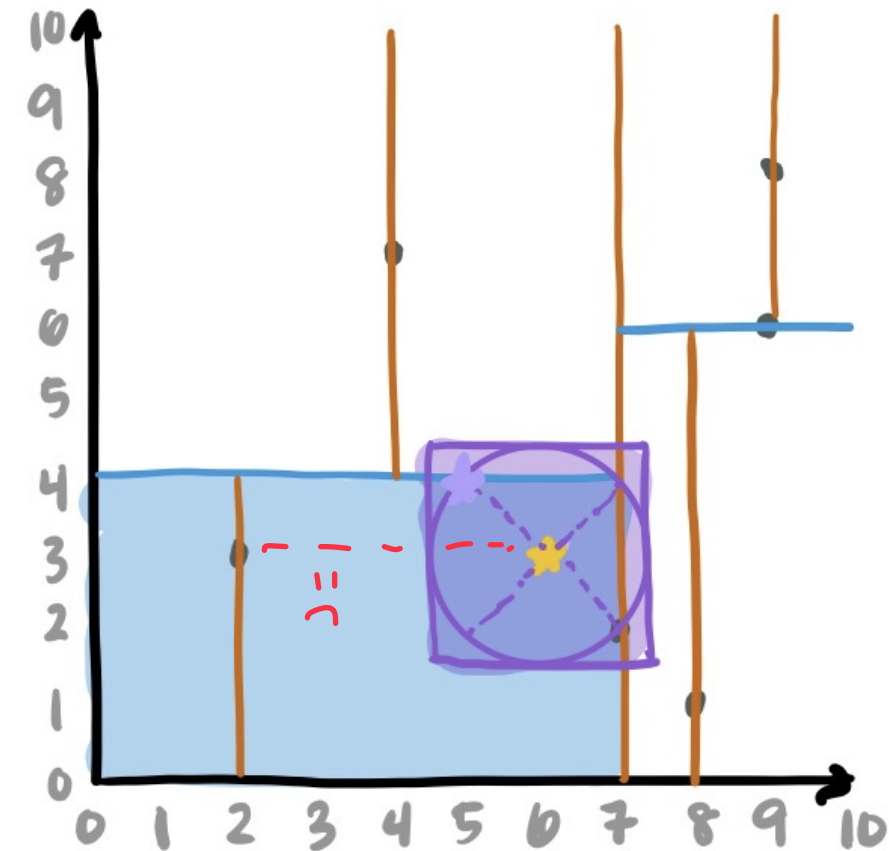
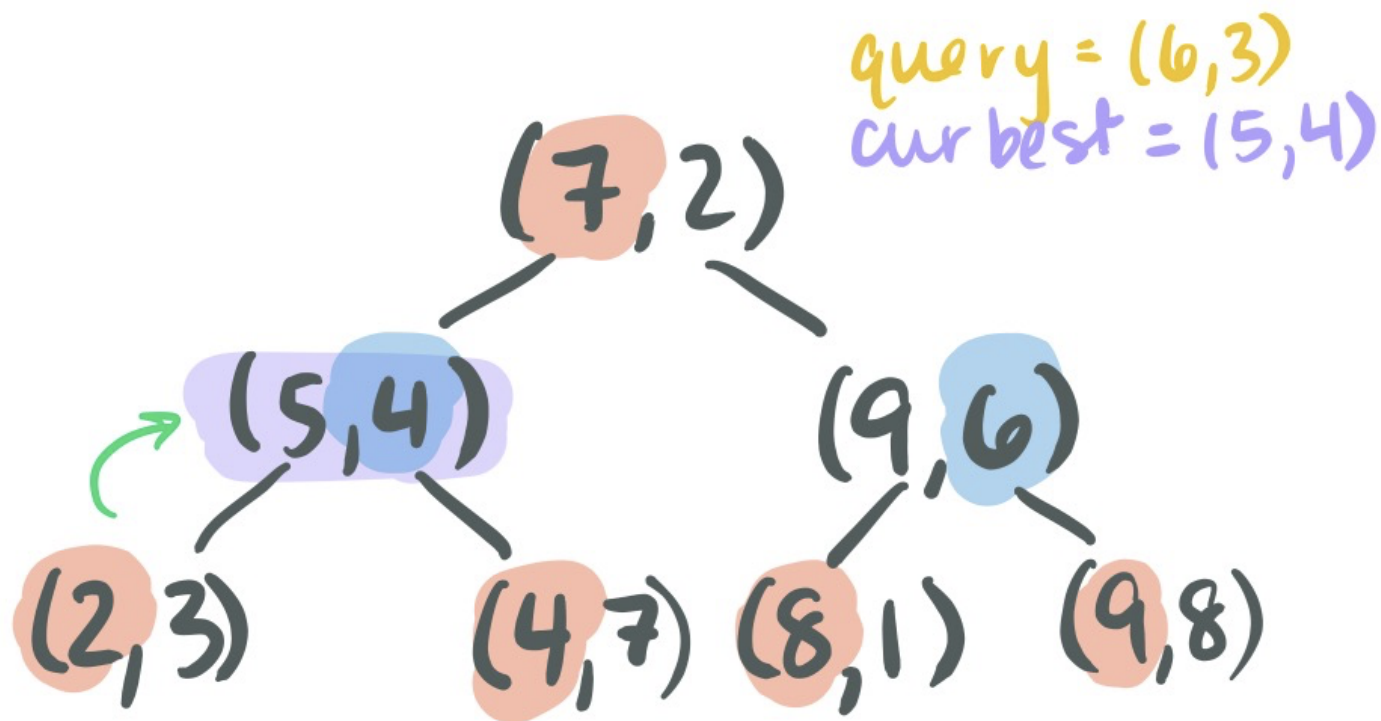
Nearest Neighbor: k-d tree

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



Nearest Neighbor: k-d tree

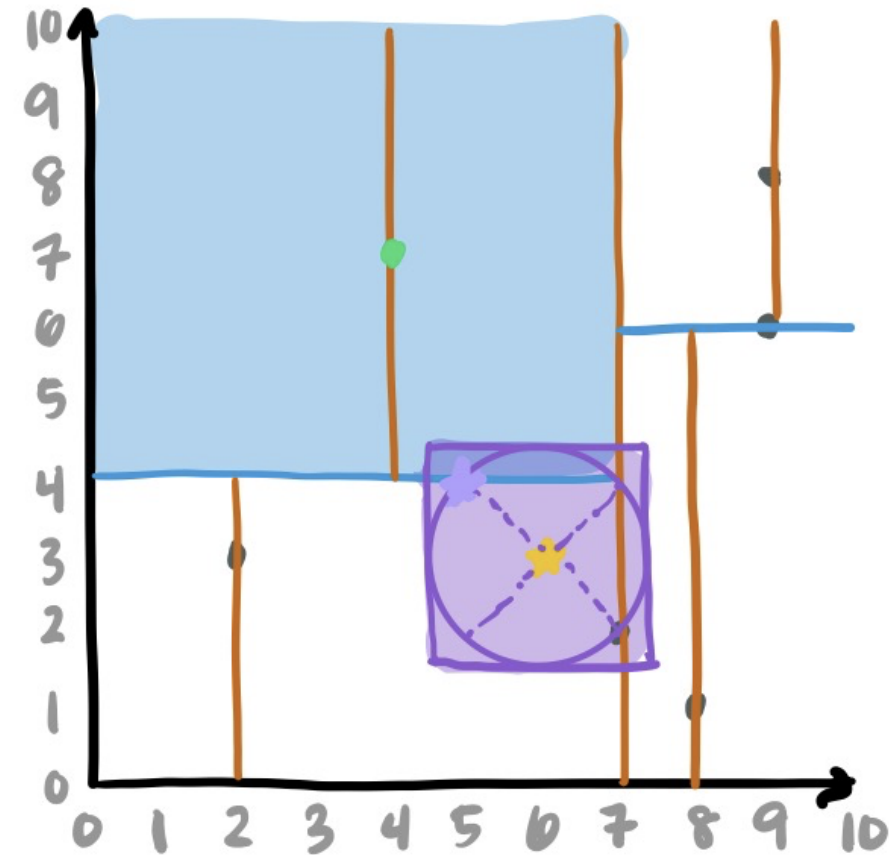
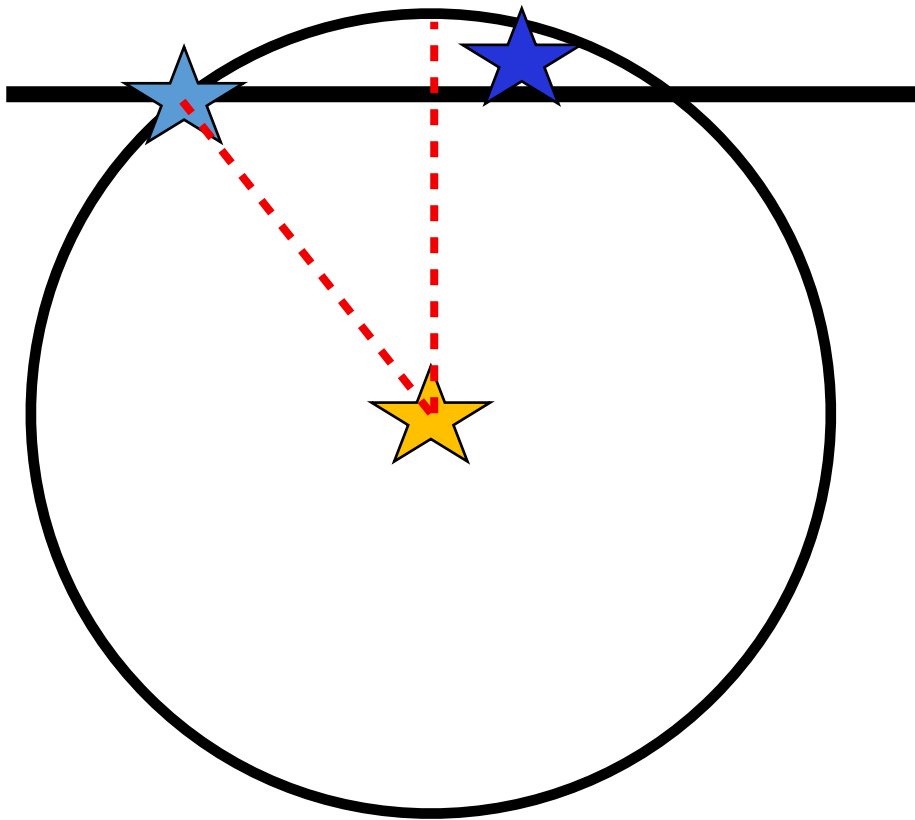
Backtracking: start recursing backwards -- store "best" possibility as you trace back



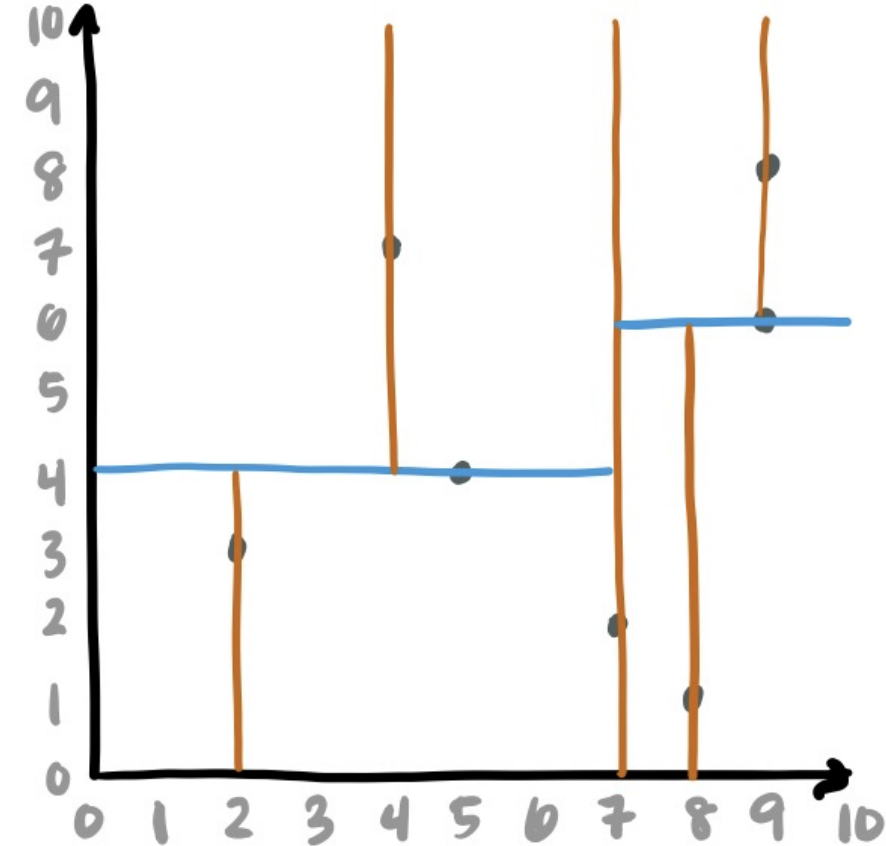
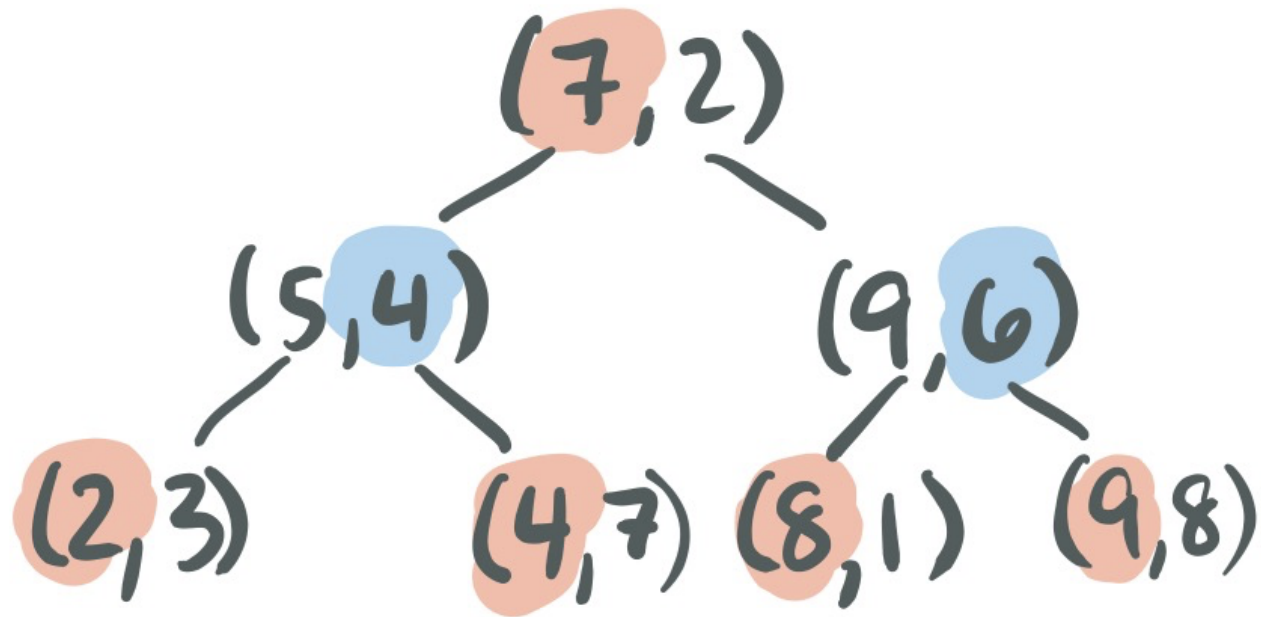
Nearest Neighbor: k-d tree

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

Potentially better point in this small area



Nearest Neighbor: k-d tree



BTree Properties

A **BTree** 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.

If $\frac{\lceil \frac{m}{2} \rceil}{+1}$ is keys
is children

All leaves in the tree are at the same level.

BTree Find

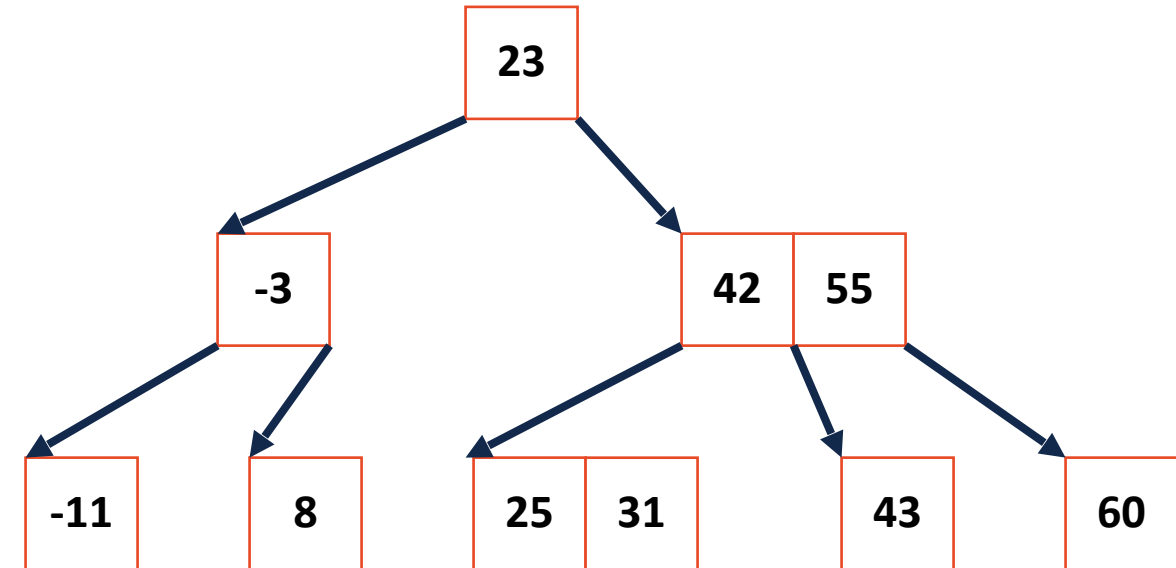
Find(7)



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

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

BTree Insertion

$$\underline{M = 5}$$

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

- 1) Create new parent node w/ median value
- 2) Split existing array into $\sim M/2$ partitions

Insert (1)

Insert (2)

Insert (3)

Insert (4)

Insert (5)

Insert (6)

Insert (7)

Insert (8)

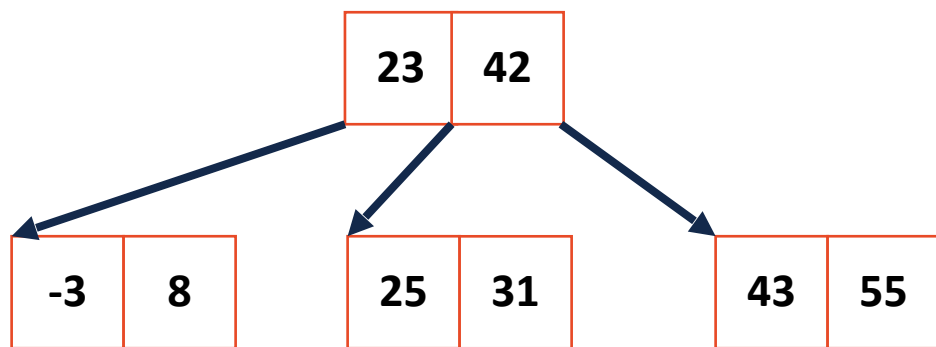
1	2	3	4	5
---	---	---	---	---

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**...

You've seen trees on previous exams...

Heap

Taking advantage of special cases in lists / arrays

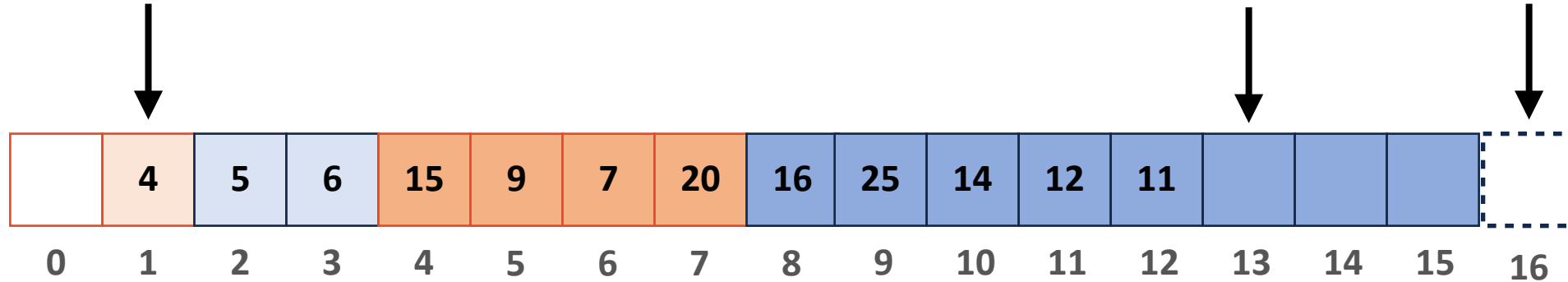
$O(1)$ lookup

$O(1)$ swap

$O(1)^*$ insertBack

Array List (Pointer implementation)

T* Start



T* Size

T* Capacity

size_t Start

size_t Capacity

Array List (Index implementation)

size_t Size

(min)Heap

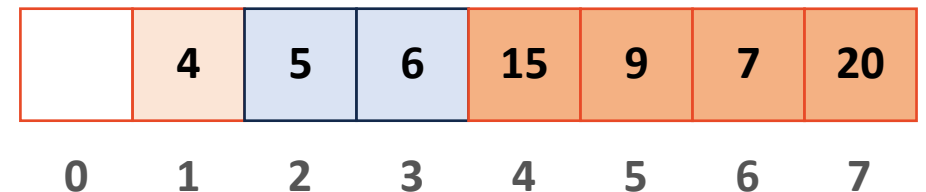
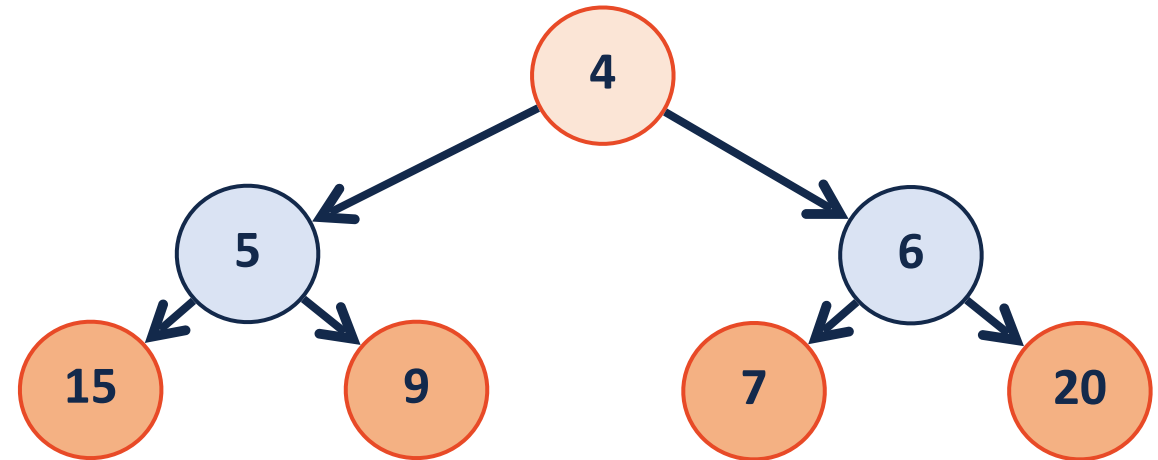
By storing as a complete tree, can avoid using pointers at all!

If index starts at 1:

`leftChild(i) : 2i`

`rightChild(i) : 2i+1`

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

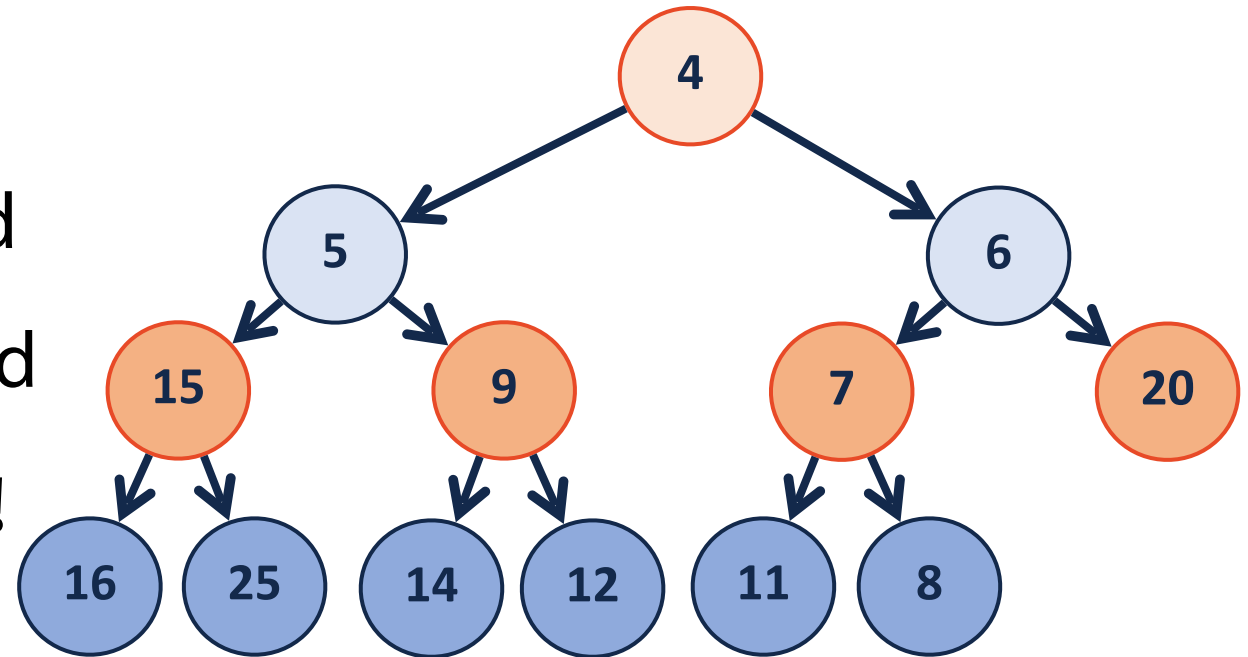


insert

Insert (2)

- 1) Insert at end of array
- 2) Check if minHeap still valid
- 3) Swap with parent if needed

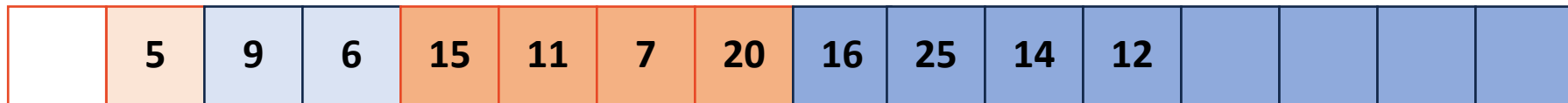
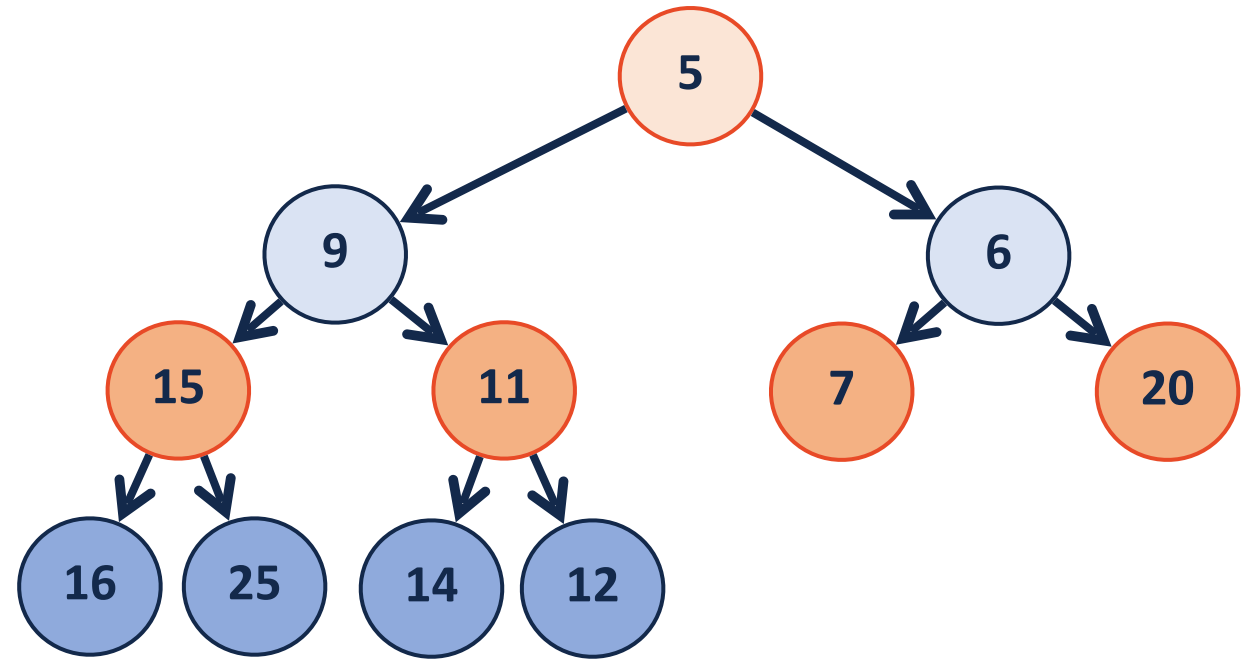
Steps 2 and 3 are recursive!



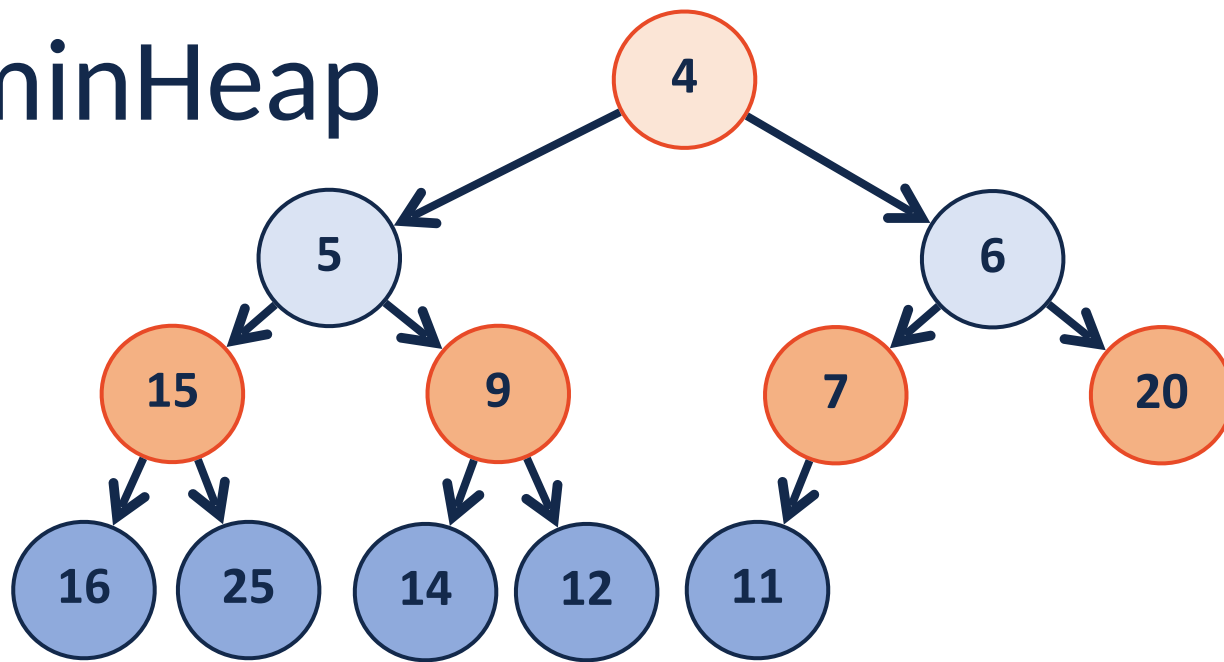
	4	5	6	15	9	7	20	16	25	14	12	11	8		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

removeMin

- 1) Swap root with last item
(and remove)
(and modify size)
- 2) HeapifyDown() root



minHeap



1. Construction

$O(n)$:)

2. Insert

$O(\log n)$

3. RemoveMin

$O(\log n)$



minHeap is a good example of tradeoffs:

Array memory locality

Fast access of min item

Not intended for random access

Some wasted space

Final thoughts on Heaps

Building a heap on different datasets is a useful exercise

You haven't been tested on heaps yet...

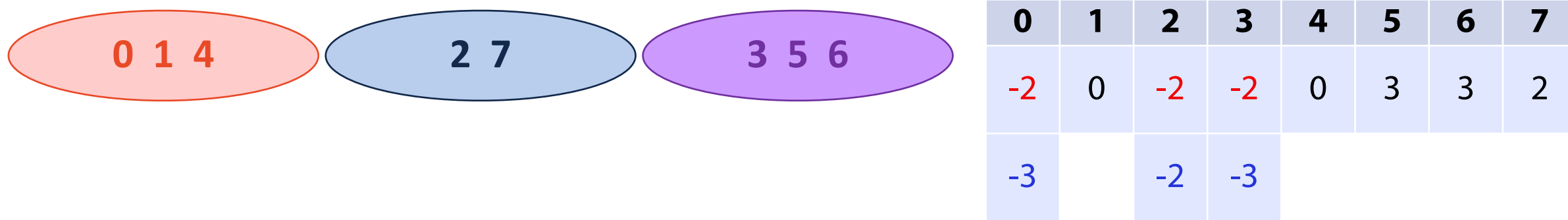


Disjoint Sets

Disjoint Set Implementation

Taking advantage of array lookup operations

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



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

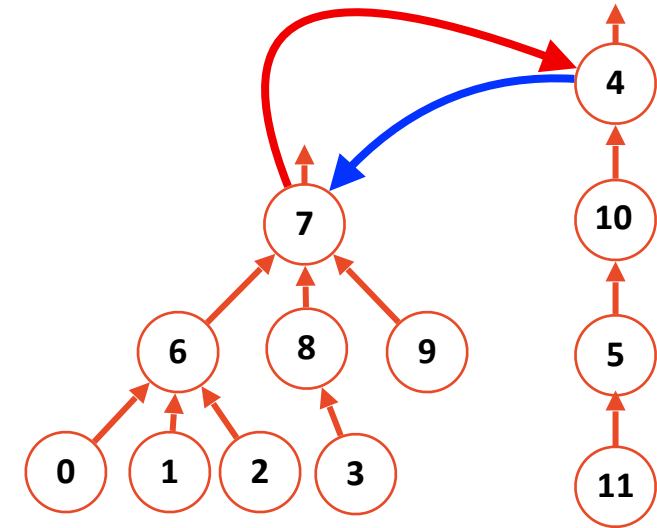
Union(k₁, k₂): Update ***smaller*** canonical item to point to larger
Update value of remaining canonical item

Disjoint Sets – Smart Union

Two $O(1)$ methods of combining two sets

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

Height 2 $\rightarrow$ Height 3



Size 4 $\rightarrow$ Size 8

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

Idea: Minimize the number of nodes that increase in height

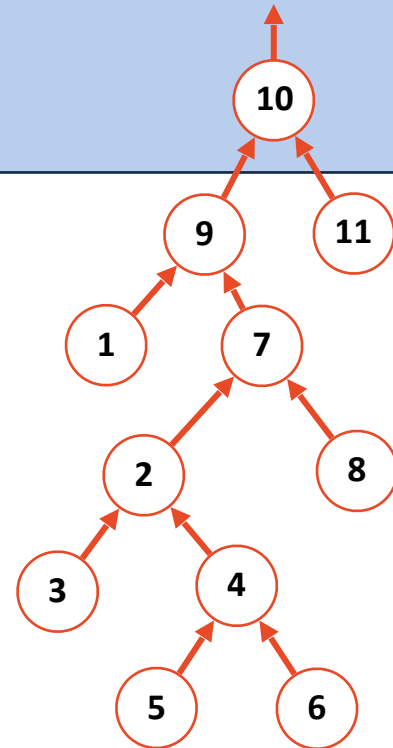
	Before Union			After Union		
Union by height	4	...	7	4	...	7
	-4		-3	-4		4
Union by size	4	...	7	4	...	7
	-4		-8	7		-12

Disjoint Sets Path Compression

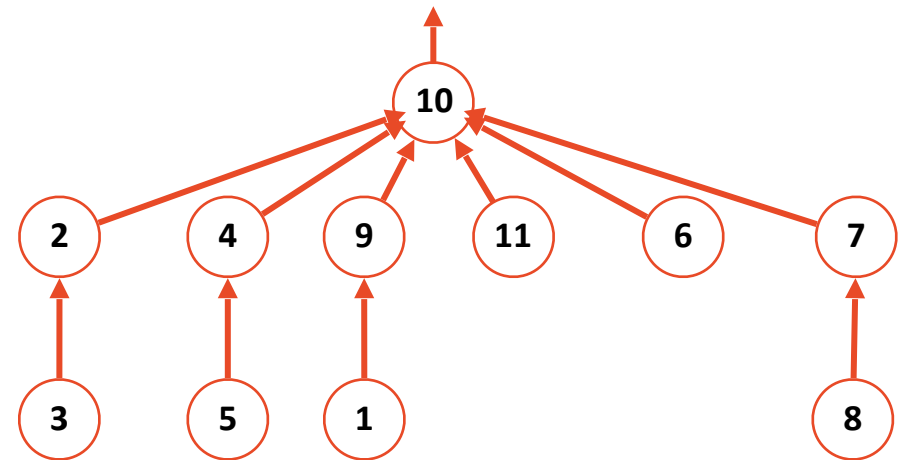
Find(6)

Minimizing number of $O(1)$ operations

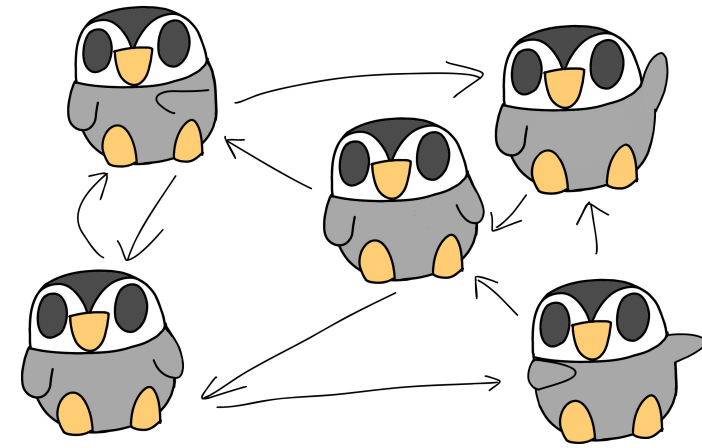
```
1 int DisjointSets::find(int i) {  
2   if ( s[i] < 0 ) { return i; }  
3   else {  
4     int root = find( s[i] );  
5     s[i] = root;  
6     return root;  
7   }  
8 }
```



Yet another benefit to array usage!

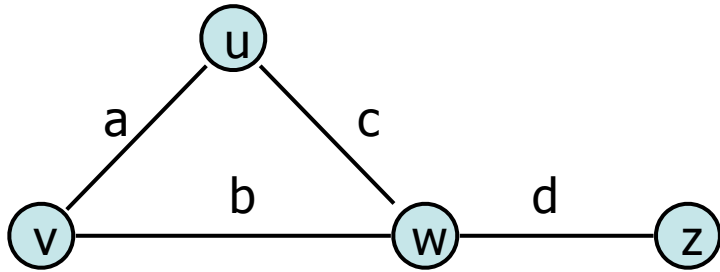


Graphs



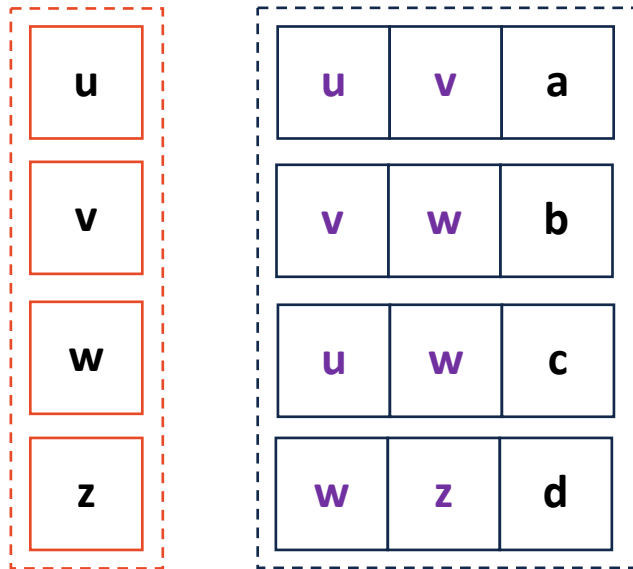
Graph Implementation: Edge List $|V| = n, |E| = m$

The equivalent of an 'unordered' data structure



Vertex Storage:

An optional list of vertices



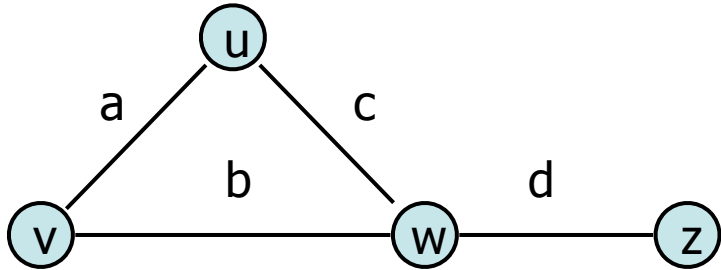
Edge Storage:

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

Edge Storage:

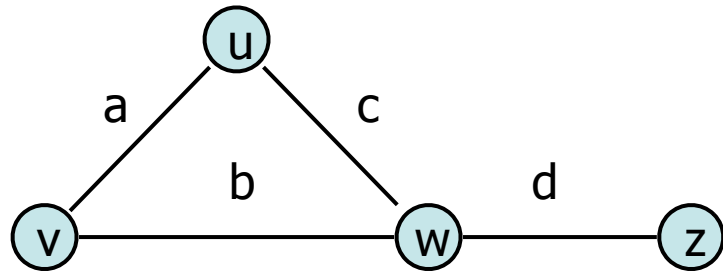
A $|V| \times |V|$ matrix of edges

Weight is stored at position (u, v)

u	0
v	1
w	2
z	3

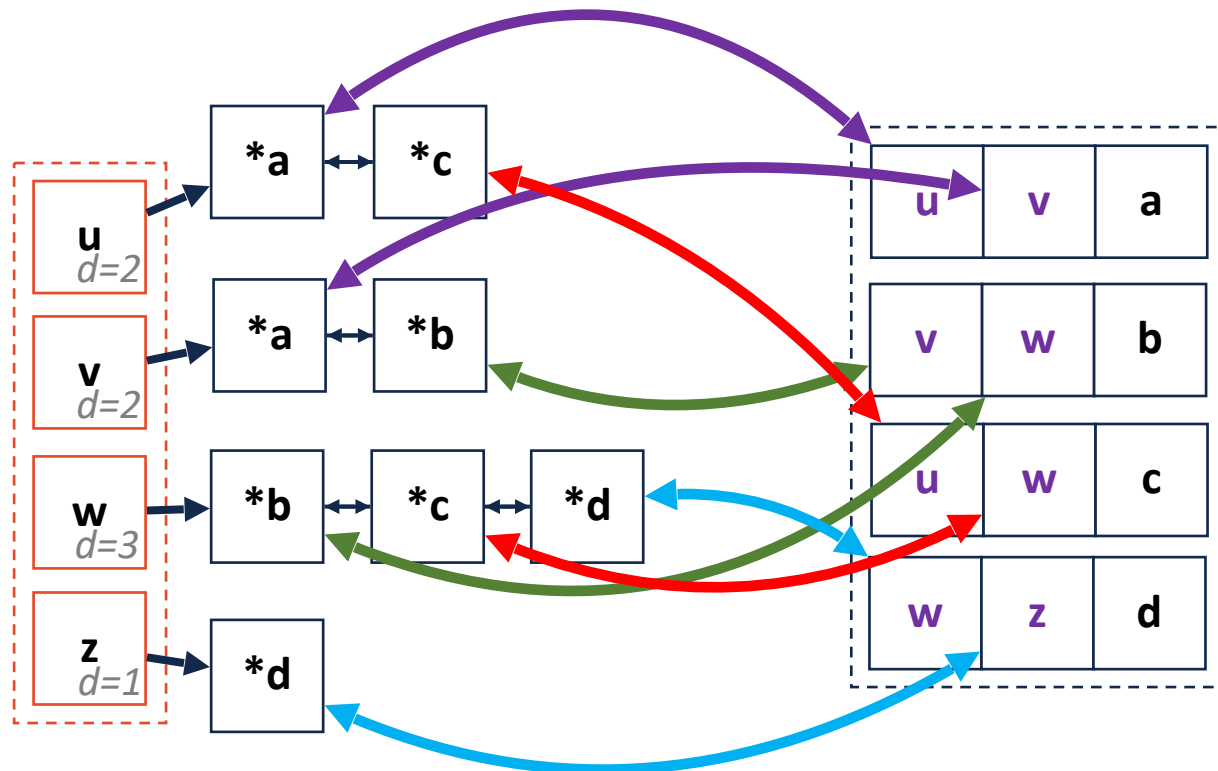
	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

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 $(v1, v2, \text{weight})$ edges
Also store pointers back to nodes

Adjacency List

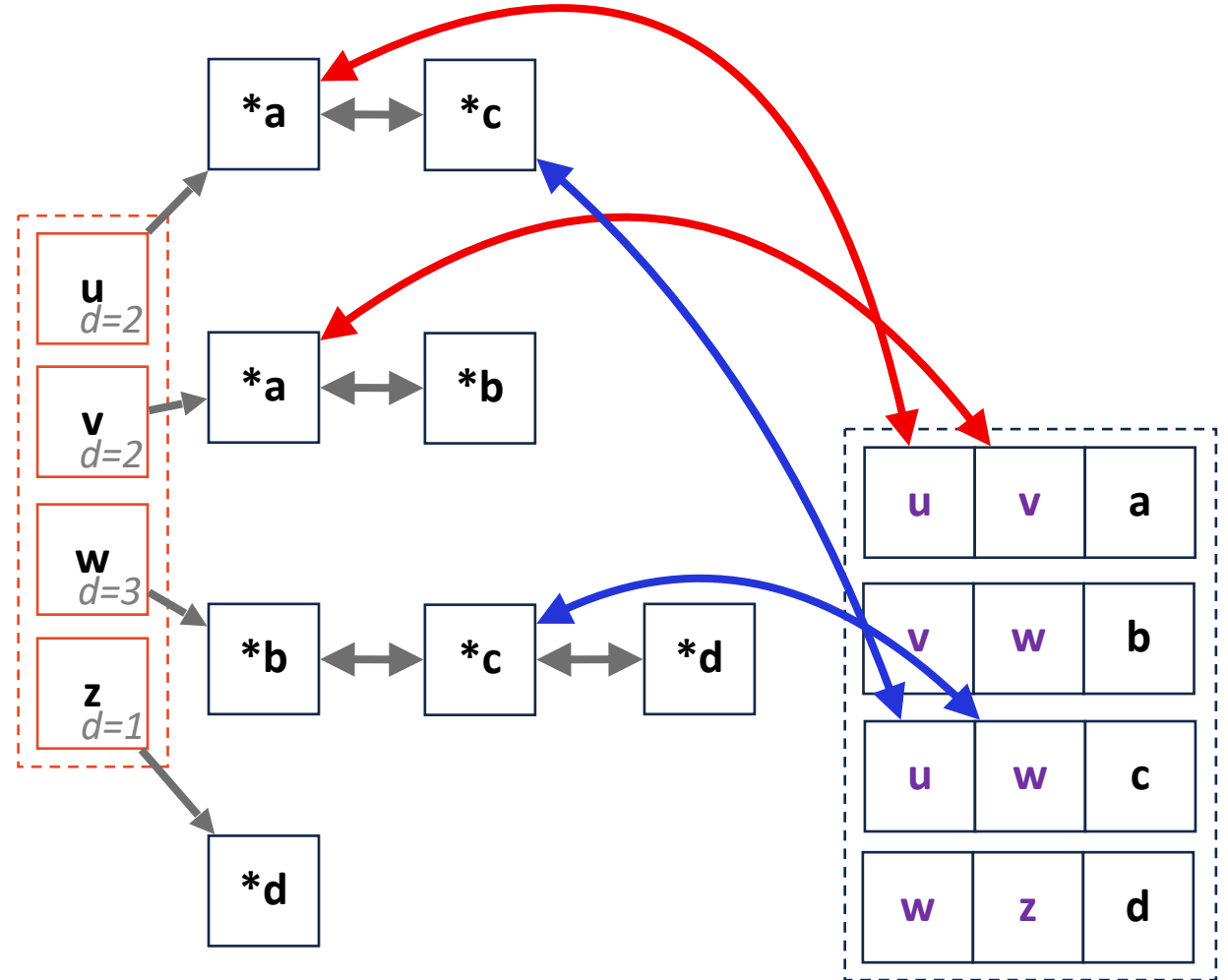
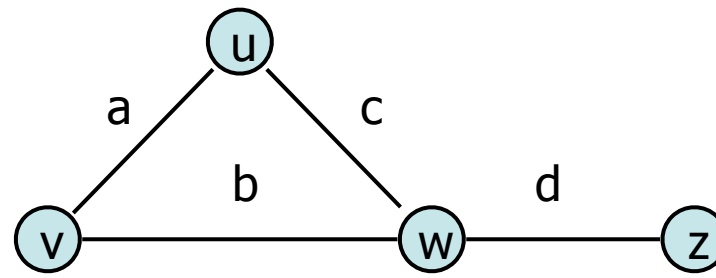
$$|V| = n, |E| = m$$

Adj List Node:

*Prev	*Edge	*Next
-------	-------	-------

Edge List:

V1	V2	Weight
*V1	*V2	



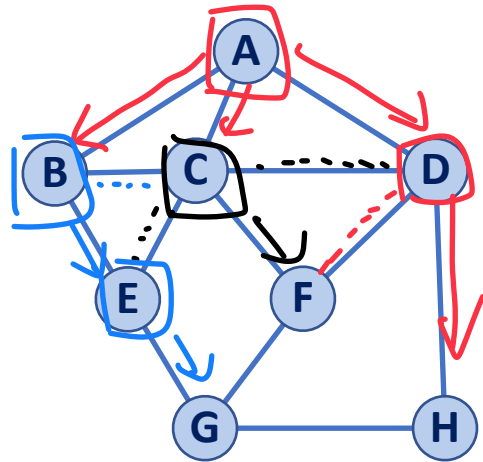
$$|V| = n, |E| = m$$



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	$\text{deg}(v)$
insertEdge(u, v)	1	1	1^*
removeEdge(u, v)	m	1	$\min(\text{deg}(u), \text{deg}(v))$
incidentEdges(v)	m	n	$\text{deg}(v)$
areAdjacent(u, v)	m	1	$\min(\text{deg}(u), \text{deg}(v))$

Traversal: BFS

- 0) Initialize
 - ↳ Queue (put start in queue)
 - ↳ Depth (start = 0)
 - ↳ Predecessor (start = -1)



- While queue not empty
 - ↳ tmp = dequeue()
 - ↳ process all children
 - ↳ Add to queue
 - ↳ Set depth (tmp.depth + 1)
 - ↳ Set prev (to tmp)

→ Discovery (New vertex)
 Cross (old vertex)

All discovery edges are in table
 All edges not in table are cross

"visited" nodes have depth/prev

depth prev			
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	2	E	E F H
H	2	D	D G

check if vertex is "new"
 Does it already have depth/prev

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

Front

(queue)

Back

Traversal: BFS

Initialize queue / depth / predecessor

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

Every edge visited twice

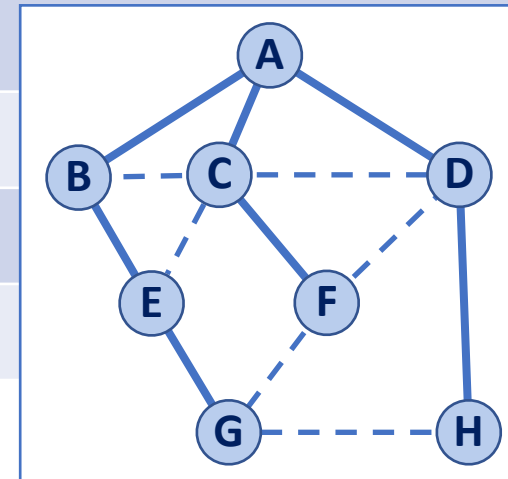
Every vertex processed once

Running time? $O(n + m)$

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

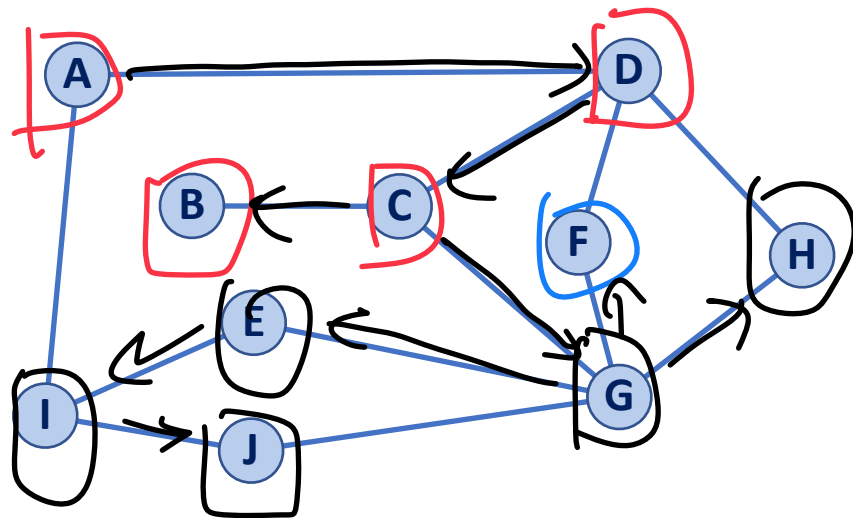


v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	1	A	A B D E F
D	1	A	A C F H
E	2	B	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G



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

Traversal: DFS



Initialize dist / pred / ~~stack~~

All dist null (start node dist 0)

All pred -1 (start node pred -1)

Stack loaded with start node

While stack not empty

tmp=stack.peek()

Process one child of tmp

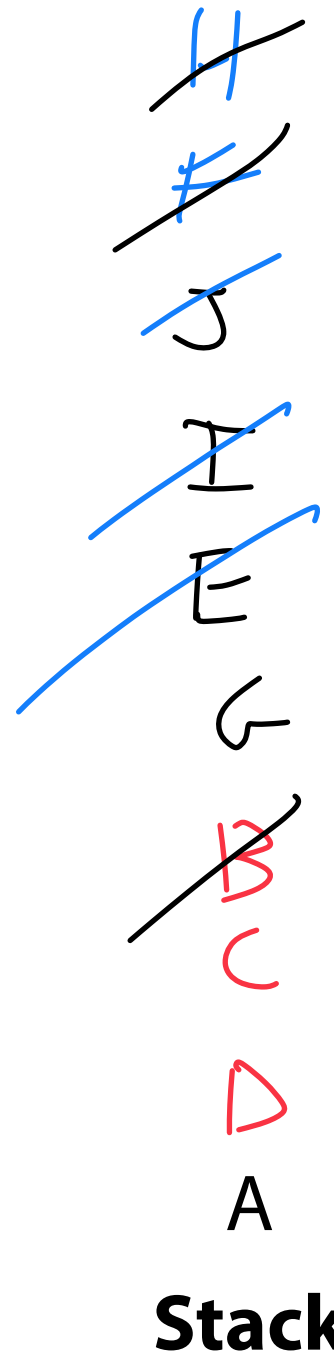
Add to stack

dist = tmp.dist + 1

pred = tmp

If no unvisited children

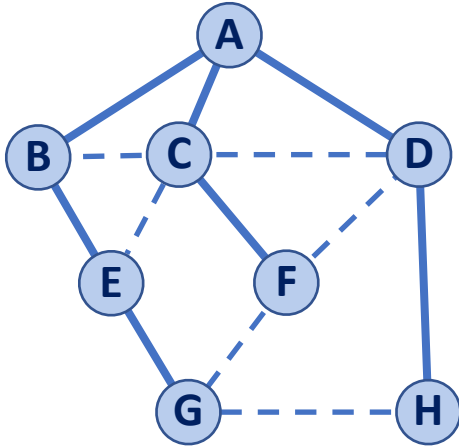
stack.pop()



Efficiency: DFS vs BFS

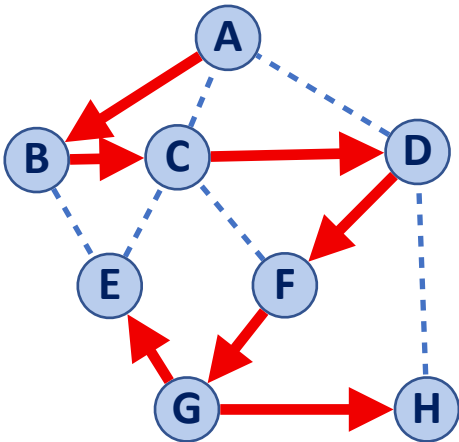
$$|V| = n, |E| = m$$

BFS: $O(n + m)$



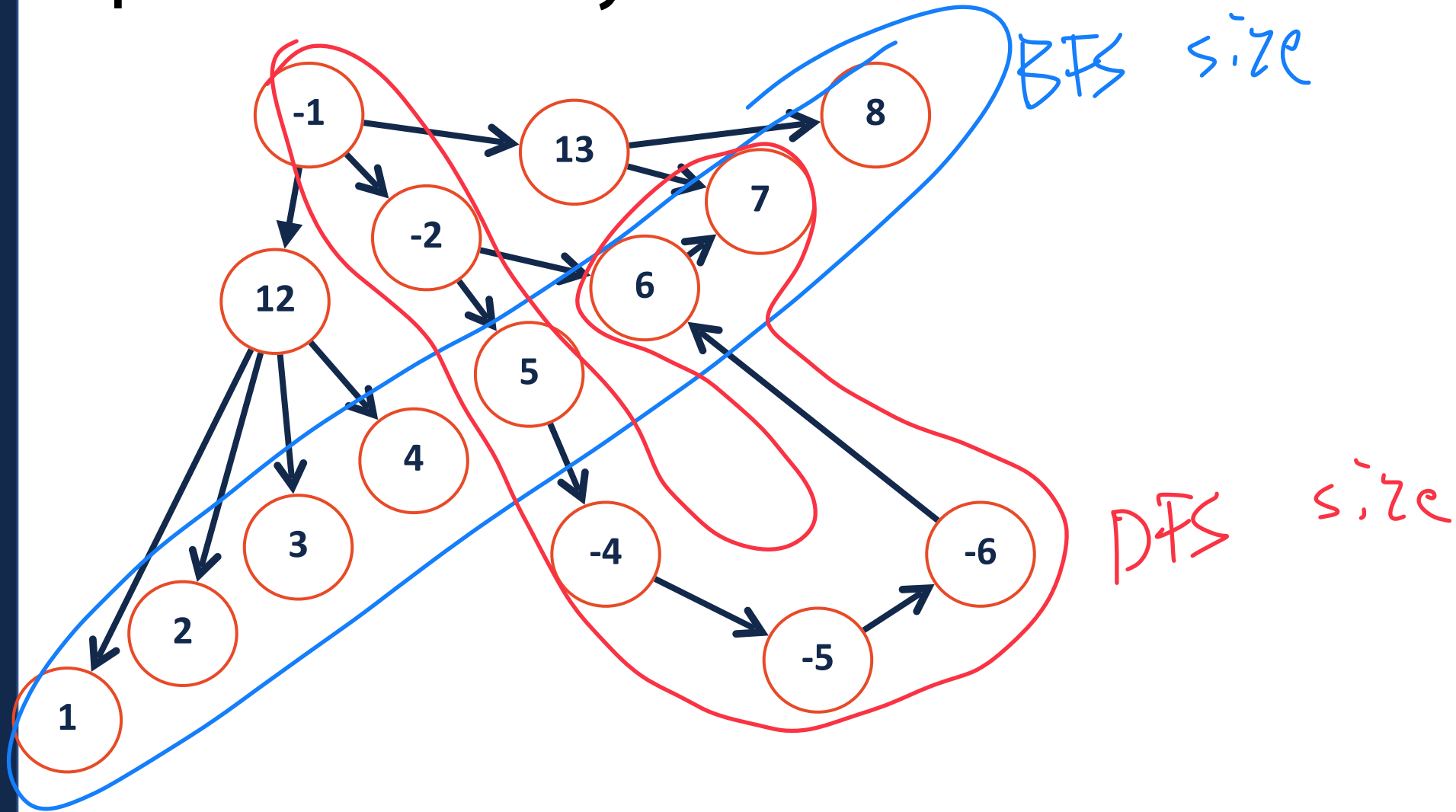
A B C D E F H G

DFS: $O(n + m)$



A B C D F G E H

Space Efficiency: DFS vs BFS



Summary: DFS and BFS

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



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

BFS

Solves unweighted MST

Solves shortest path

Solves cycle detection

Memory bounded by width

DFS

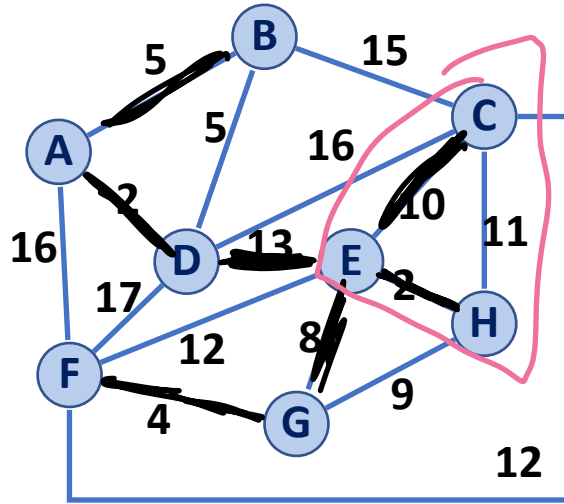
Solves unweighted MST

Solves cycle detection

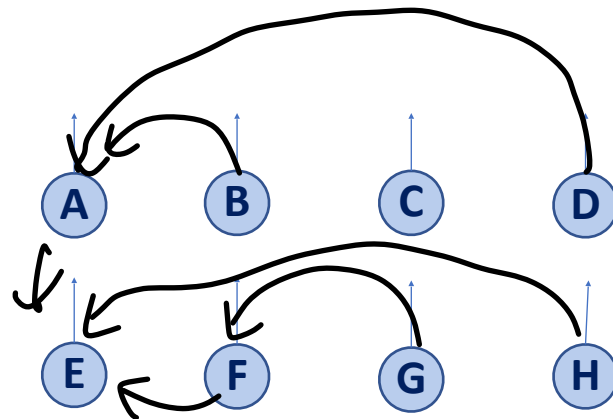
Memory bounded by longest path

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)	



At time step here



November 3rd

- 1) Build a **priority queue** on edges
A minheap or *A sorted array*
- 2) Build a **disjoint set** on vertices
All vertices start as their own set
- 3) Loop through min edges
If edge connects two disjoint sets
update *Union sets and record edge in MST*
- 4) Stop when:
we have MST
N-1 edges recorded
Only a single disjoint set remains

Kruskal's Algorithm

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

What is the Big O?

2 — 4: $O(n)$

6 — 7: Heap: $O(m)$
 Sorted List: $O(m \log m)$

11: $m \times \langle 12-17 \rangle$

12—17: Heap: $O(\log m)$
 Sorted List: $O(1)$

```
1  KruskalMST(G) :
2      DisjointSets forest
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
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
```

Disjoint set we treat as $O(1)$ b/c path compression w/ smart union

Kruskal's Algorithm

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

Both result in $m + m \log m$

Why is heap good?

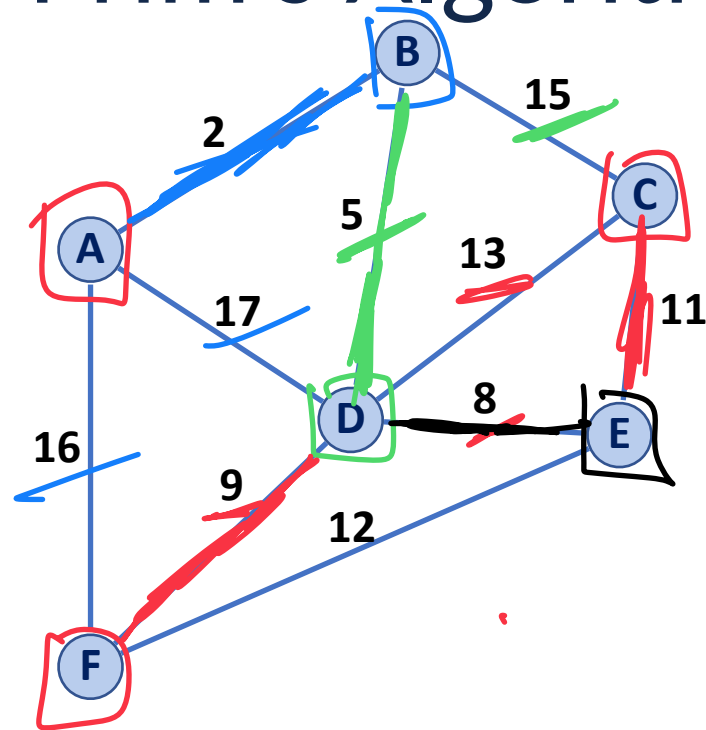
If edge weights can change!

Why is sorted array good?

Sorted array not destroyed and can be useful in other algorithms!

[illegible]

Prim's Algorithm



A	B	C	D	E	F
0, -	1, B	1, B	1, B	1, B	1, B
2, A	2, A	15, B	17, A	8, D	10, A

13, D
11, F
5, B

```

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



Prim's Big O

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

7 — 9: $O(n)$

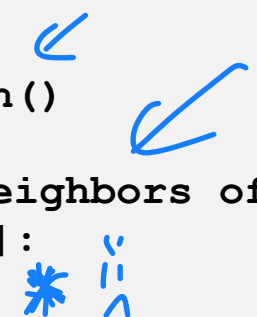
12—14:

MinHeap: $O(n)$

Unsorted Array: $O(1)$

16—22: Complicated!

```
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
```

Hand-drawn blue arrows and a star pointing to lines 16-22 of the code. One arrow points to line 16, another to line 17, and a third to line 19. A blue star is drawn next to line 21.

Depends on choice of **PriorityQueue** (MinHeap vs Unsorted Array)

Depends on choice of **Graph** (Adjacency Matrix vs Adjacency List)

Prim's Algorithm

Sparse Graph: $m \sim n$

Adj List Heap best

Dense Graph: $m \sim n^2$

Unsorted Array best

```
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	$O(n^2)$	$O(n^2)$

MST Algorithm Runtime:

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

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

Sparse Graph: $m \sim n$

Both are $n \log n$

Dense Graph: $m \sim n^2$

Both are $n^2 \log n$

Dijkstra's Algorithm (SSSP)

November 5th

$O(m + n \log n)$

What is the running time of Dijkstra's Algorithm? The same as Prim's!

6-9: $O(n)$

11-12: $O(n)$

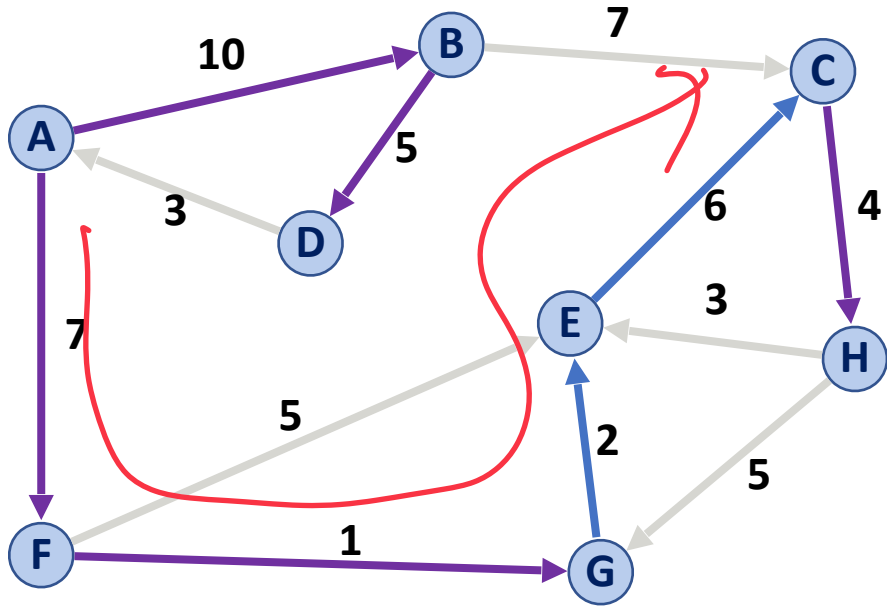
15: repeat below n x

16-22: $O(\log n)$

[w/ Fib Heap $O(1)$ updates]

```
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] = u
22
23   return T
```

Dijkstra's Algorithm (SSSP)



This is not MST!

DijkstraSSSP(G, s):

```

6  foreach (Vertex  $v$  :  $G.vertices()$ ):
7       $d[v] = +\infty$ 
8       $p[v] = \text{NULL}$ 
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  $\text{cost}(u, v) + d[u] < d[v]$ :
20               $d[v] = \text{cost}(u, v) + d[u]$ 
21               $p[v] = u$ 
    
```

A	B	C	D	E	F	G	H
--	A	E	B	G	A	F	C
0	10	16	15	10	7	8	20

This solves shortest path



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: $O(n^2)$

```
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] = u
22
23   return T
```


Floyd-Warshall Algorithm

Running time? $O(n^3)$

Easy to code / multi-threadable

Can handle negative weights!

```
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 u : G):  
13      foreach (Vertex v : G):  
14          foreach (Vertex w : G):  
15              if d[u, v] > d[u, w] + d[w, v]:  
16                  d[u, v] = d[u, w] + d[w, v]
```

Final thoughts on Graphs

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

You've seen graph questions on other exams:

- 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

November 10th

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.

↳ What is the expected dice roll value

"I roll two dice"

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

$$E[X] = \sum_{x \in \Omega} \text{Prob of event} \cdot \text{Value of event} \cdot x$$

two dice

$$\frac{1}{36} (1+1) + \frac{2}{36} (1+2) + \dots$$

Prob of one dice:

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6}$$

$E[\text{one dice roll}]$

$= 3.5$

$$E[\text{two dice rolls}] = 2E[1]$$

Probabilistic Data Structures

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

Randomized data structures rely on **expected** performance

Randomized data structures 'cheat' tradeoffs!

Randomized Algorithms

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

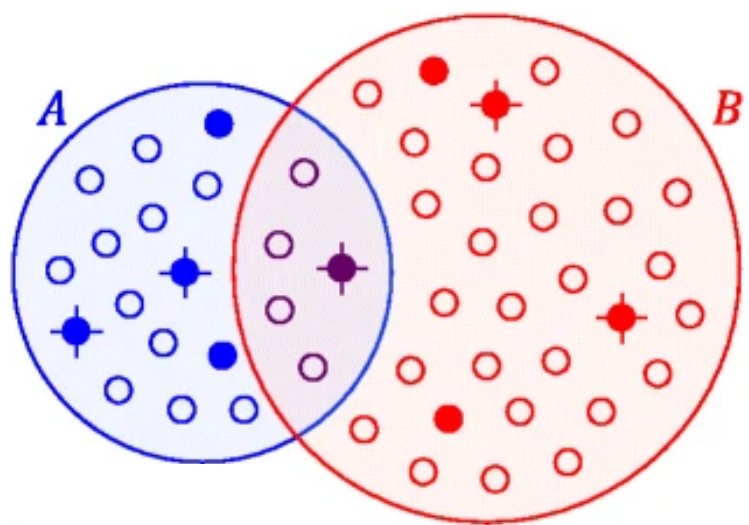
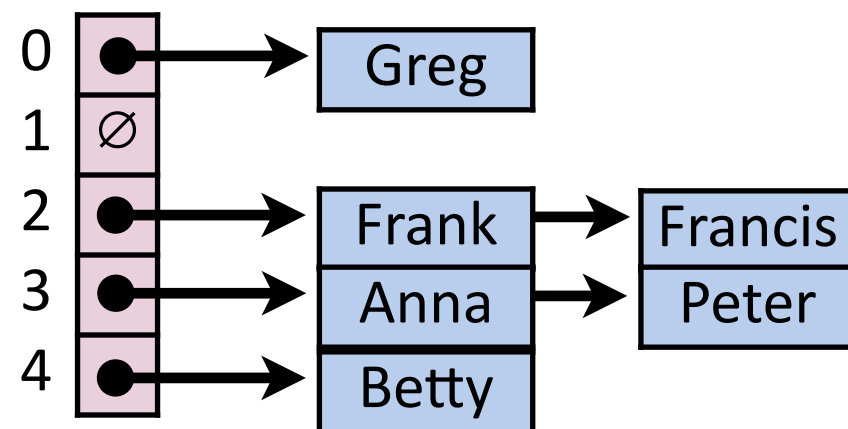
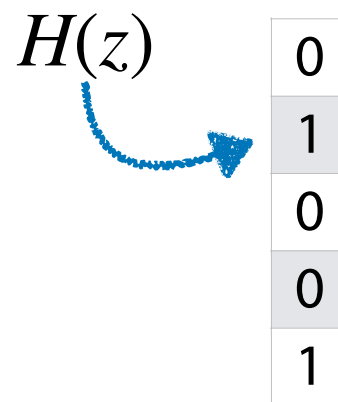


Figure from Ondov et al 2016



$H(x)$	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):

```
1 Dictionary<KeyType, ValueType> d;  
2 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

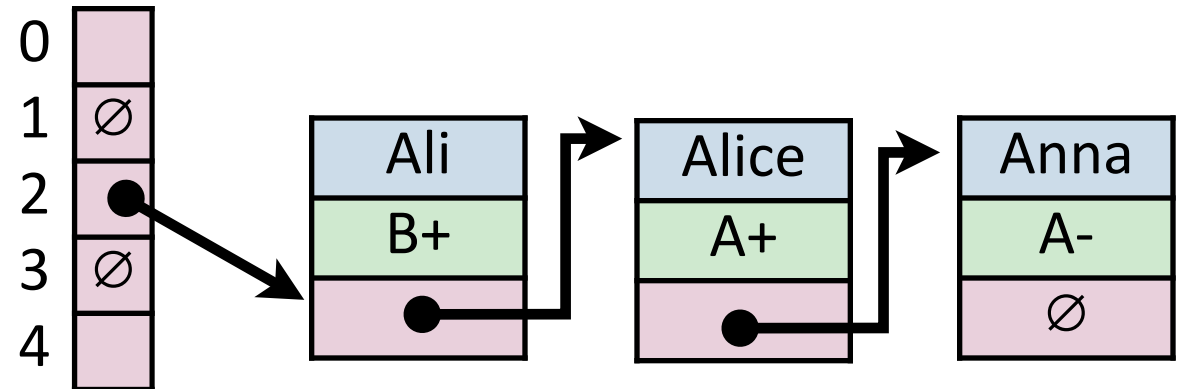
Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

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

Such as a linked list

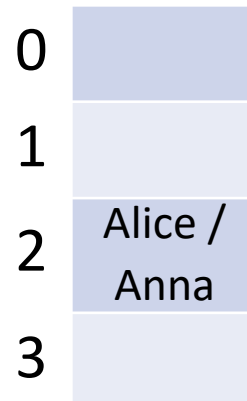
Resolve collisions by adding to list



- **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



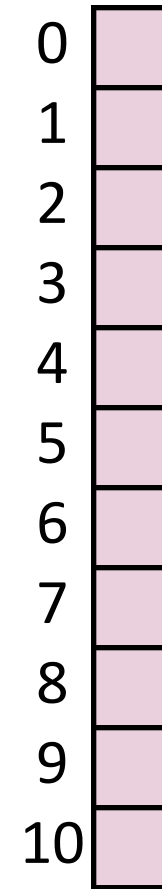
Under SUHA, a hash table of size m and n elements:

$$\alpha = n/m$$

Find runs in: $O(1+\alpha)$.

Insert runs in: $O(1)$.

Remove runs in: $O(1+\alpha)$.



Collision Handling: Linear Probing

$S = \{ 16, 8, 4, 13, 29, 11, 22 \}$ $|S| = n$

$h(k, i) = (k + i) \% 7$ $|\text{Array}| = m$

0	22
1	8
2	16
3	29
4	4
5	11
6	13

find(29)

1) Hash the input key [$h(29)=1$]

2) Look at hash value (address) position

If present, return (k, v)

If not look at **next available space**

Stop when:

1) We find the object we are looking for

2) We have searched every position in the array

3) We find a blank space

Running Times (Expectation under SUHA)



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

$$\text{insert: } \frac{1}{1}.$$

$$\text{find/ remove: } \frac{1 + \alpha}{1}.$$

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

$$\text{insert: } \frac{1}{1 - \alpha}.$$

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

Observe:

* - **As α increases:**

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

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

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

runtime

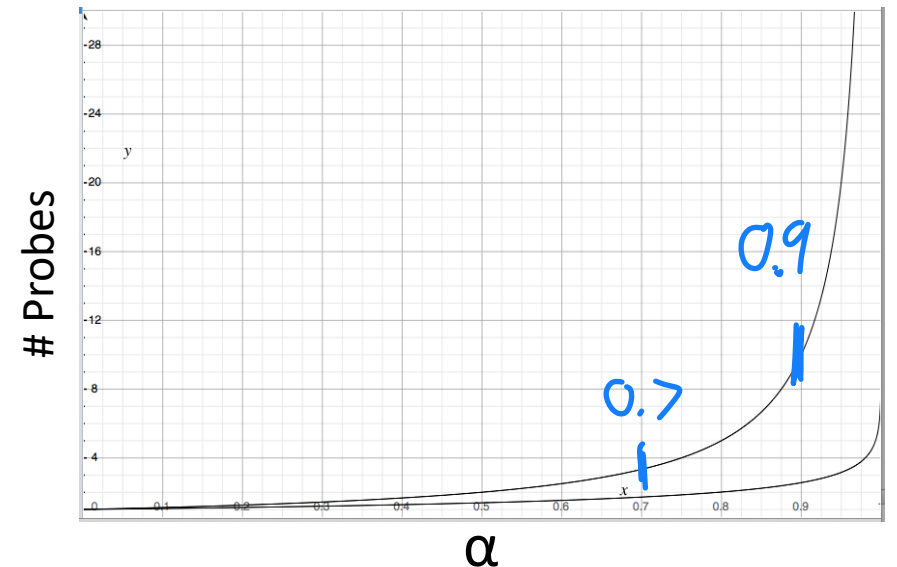
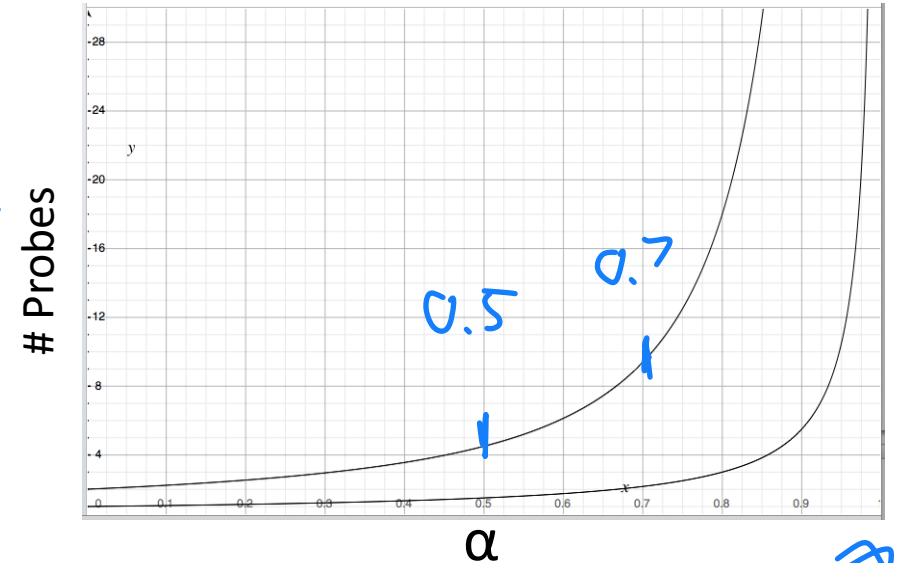
Double Hashing:

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

runtime

When do we resize?

Linear $\sim 0.7 - 0.8$
Double $\sim 0.7 - 0.9$



Running Times (Tradeoff Highlights)



	Hash Table	AVL	Linked List
Find	Expectation*: $O(1)^{***}$ Worst Case: $O(n)$	$O(\log n)$	$O(n)$
Insert	Expectation*: $O(1)^{***}$ Worst Case: $O(n)$ Separate Chaining: $O(1)$	$O(\log n)$	$O(1)$
Storage Space	$O(n)$	$O(n)$	$O(n)$

Bloom Filter

A probabilistic data structure storing a set of values

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

Insert / Find runs in: $\underline{O(k) / O(1)}$

Delete is not possible (yet)!

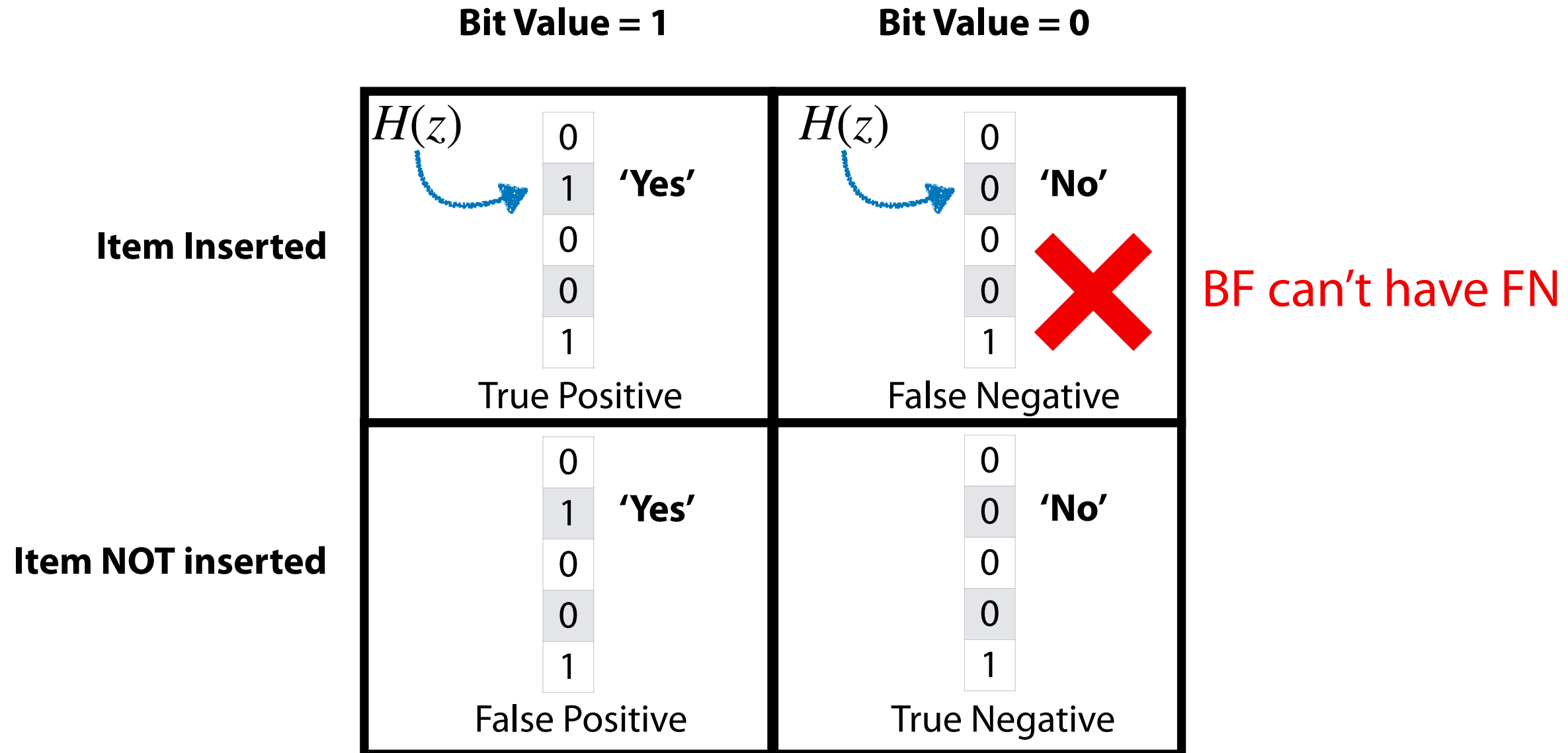


$$H = \{h_1, h_2, \dots, h_k\}$$



0
0
1
0
0
1
0
1
0
0

Probabilistic Accuracy in a Bloom Filter



Bloom Filters



A probabilistic data structure storing a set of values

$h_{\{1,2,3,\dots,k\}}$

Has three key properties:

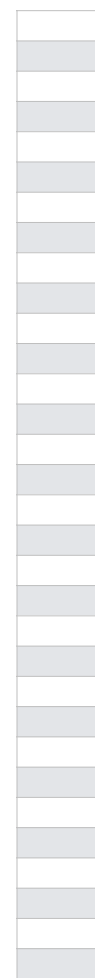
k , number of hash functions

n , expected number of insertions

m , filter size in bits

Expected false positive rate: $\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{\frac{-nk}{m}}\right)^k$

Optimal accuracy when: $k^* = \ln 2 \cdot \frac{m}{n}$



Bloom Filter: Error Rate

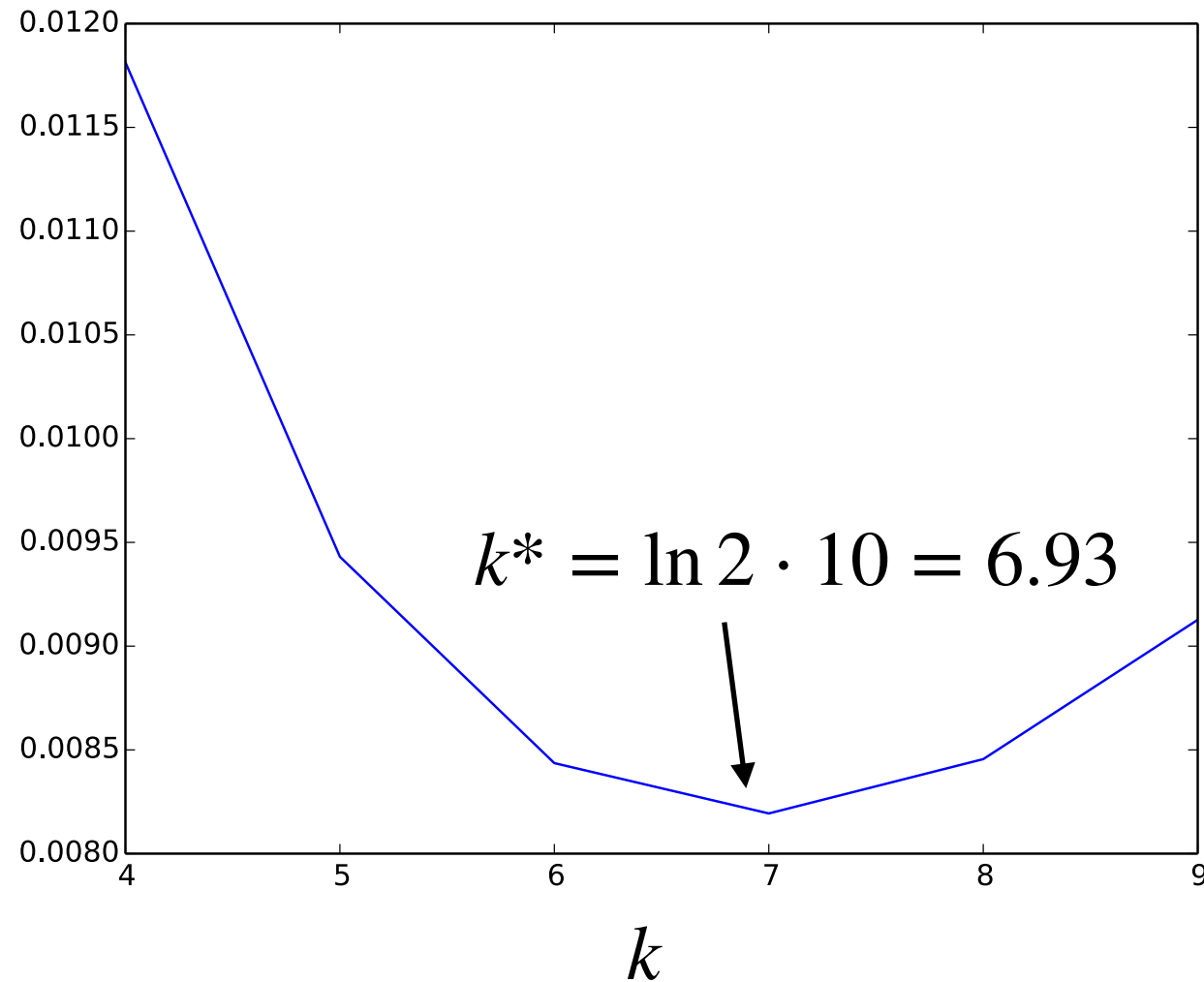


Not enough random trials

$$m/n = 10$$

BF is too saturated

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



Cardinality Estimation



Let $\text{min} = 95$. 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

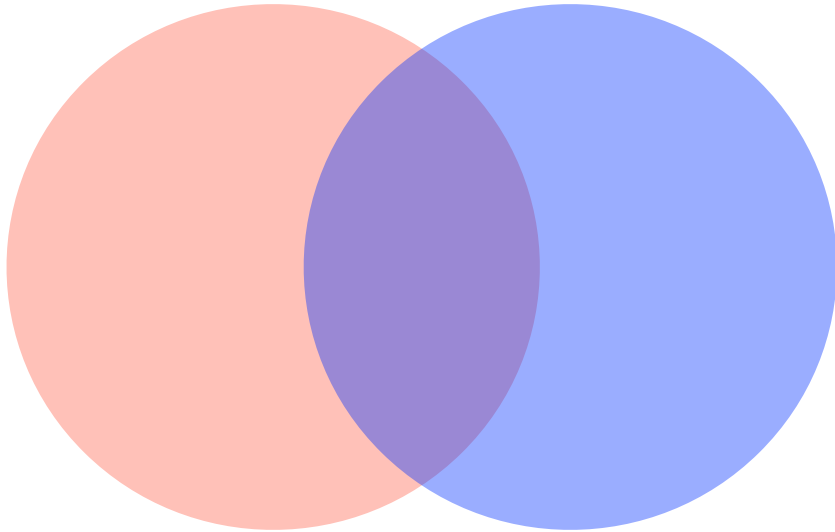
Assuming our first 'partition' is about average: $95 \approx 1000/(N + 1)$

$$N + 1 \approx 10.5$$

$$N \approx 9.5$$

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!

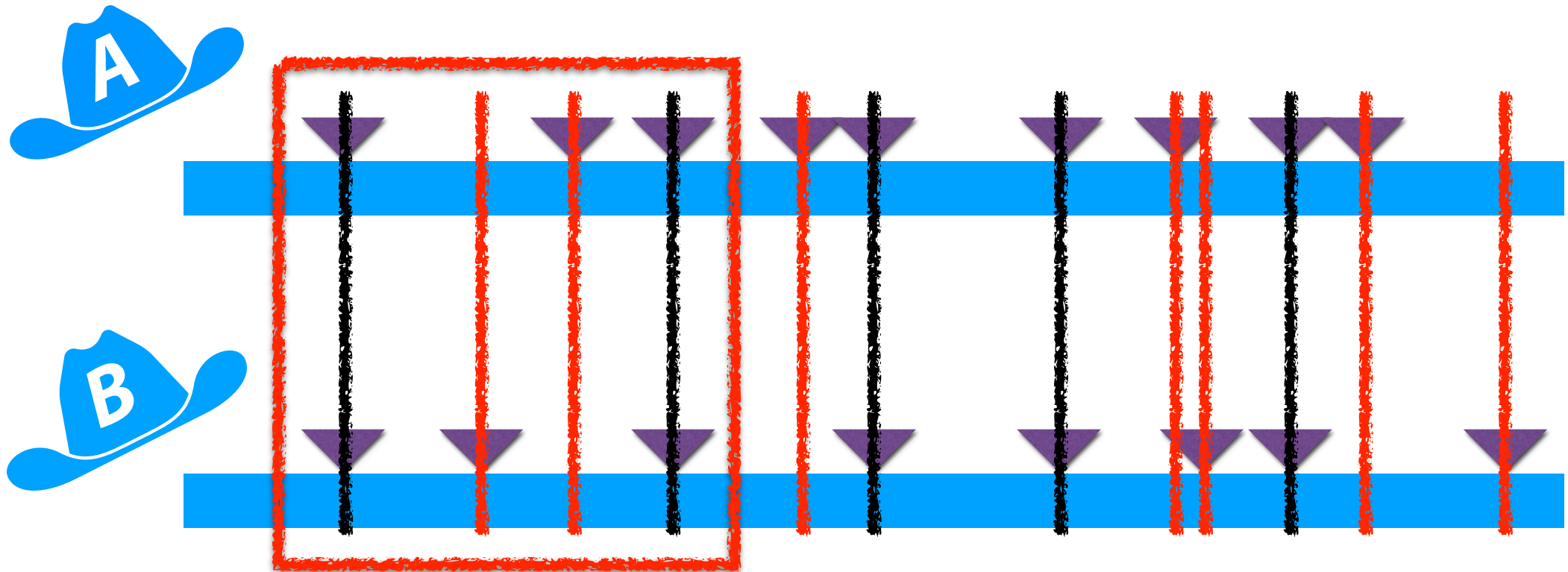
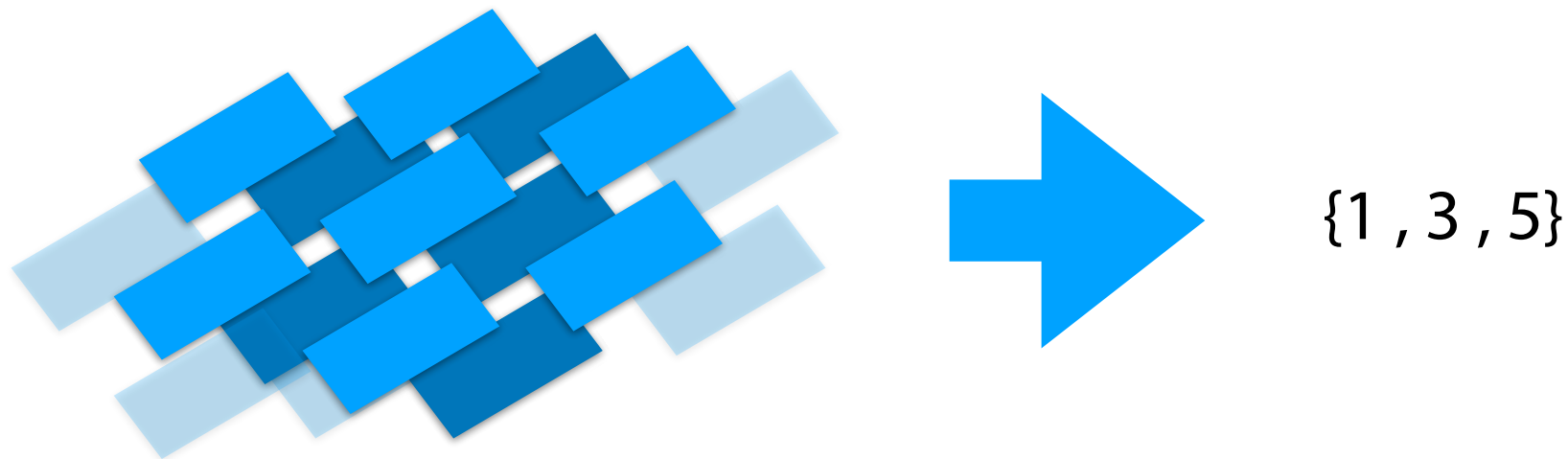


Image inspired by: Ondov B, Starrett G, Sappington A, Kostic A, Koren S, Buck CB, Phillippy AM. **Mash Screen: high-throughput sequence containment estimation for genome discovery.** *Genome Biol* 20, 232 (2019)

MinHash Sketch



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



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

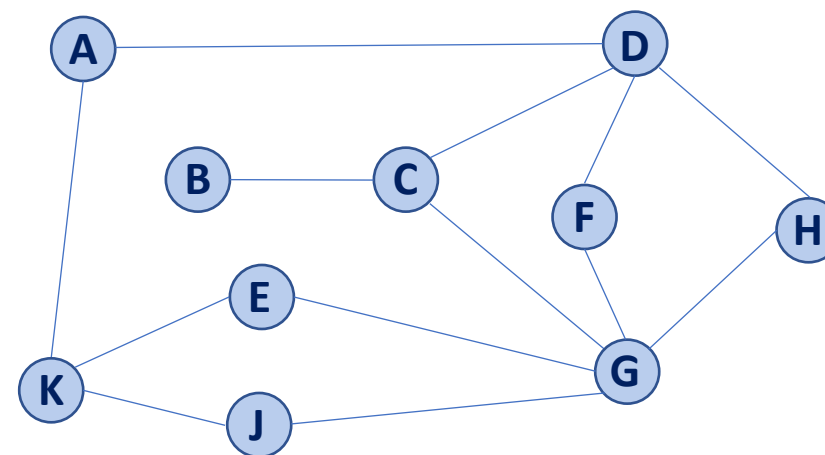
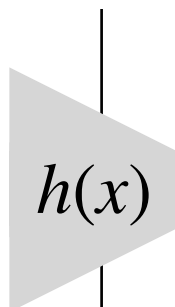
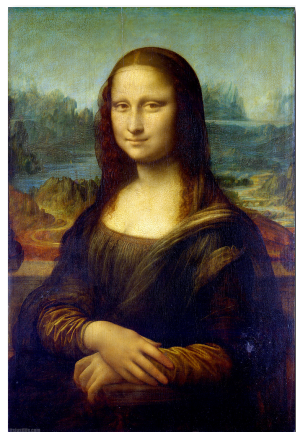
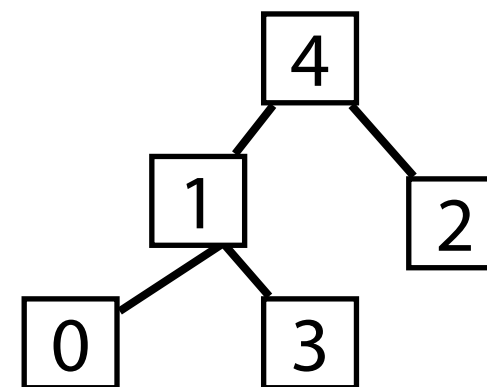
1. Cardinality Estimation (Using the k-th minimum hash value)
2. Set Similarity (Using all k-min hash values)



Questions?

CS 225 — Course Goals

Understand foundational data structures and algorithms

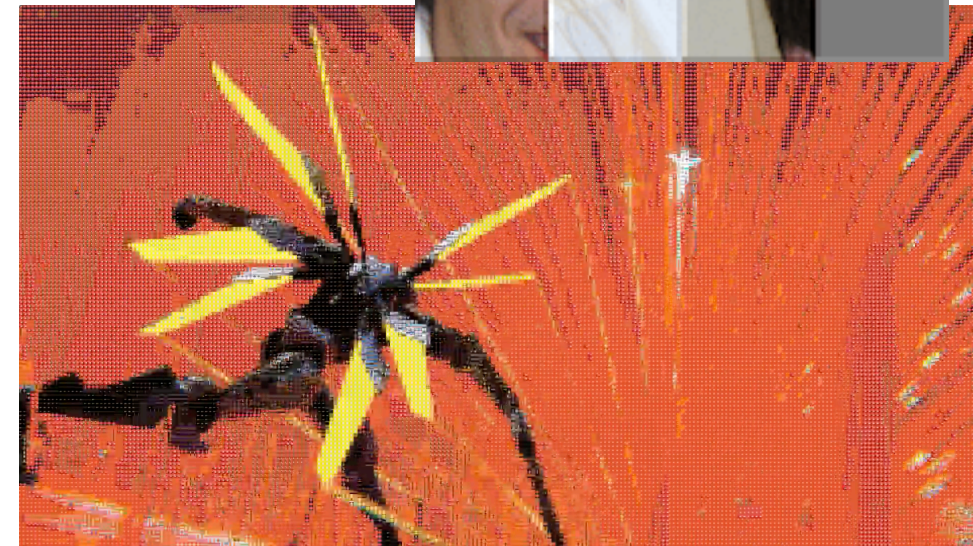
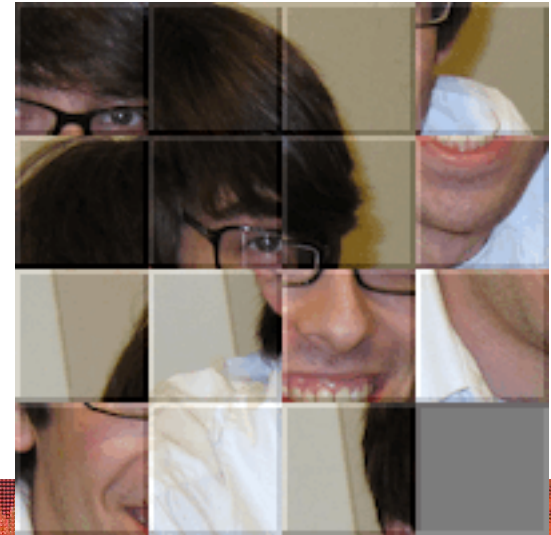


CS 225 — Course Goals

Justify appropriate algorithms for complex problems

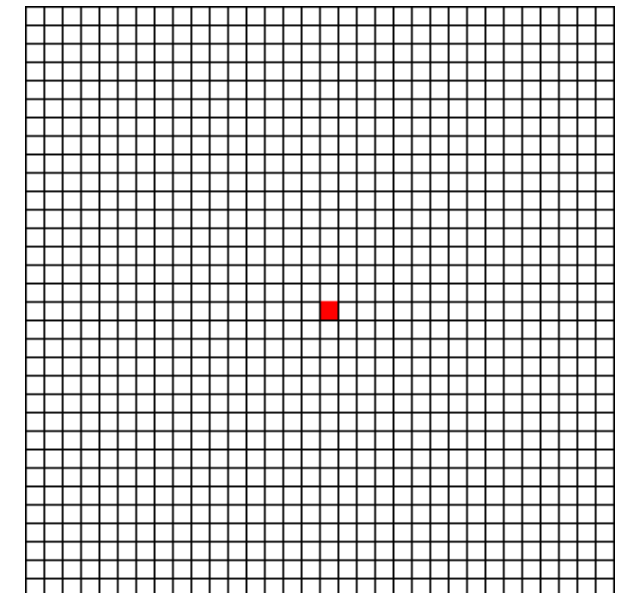
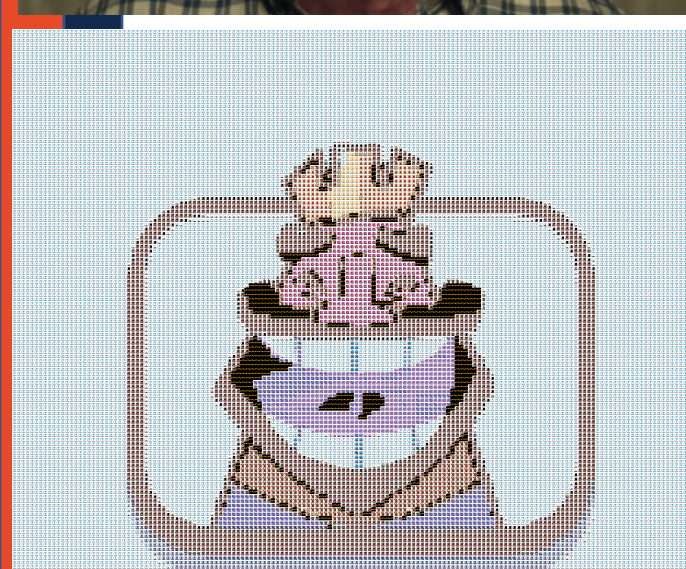
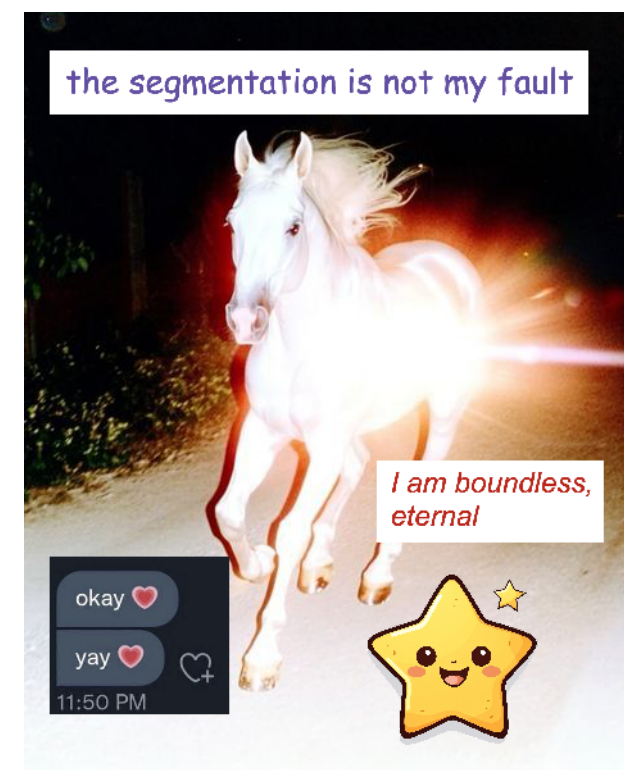
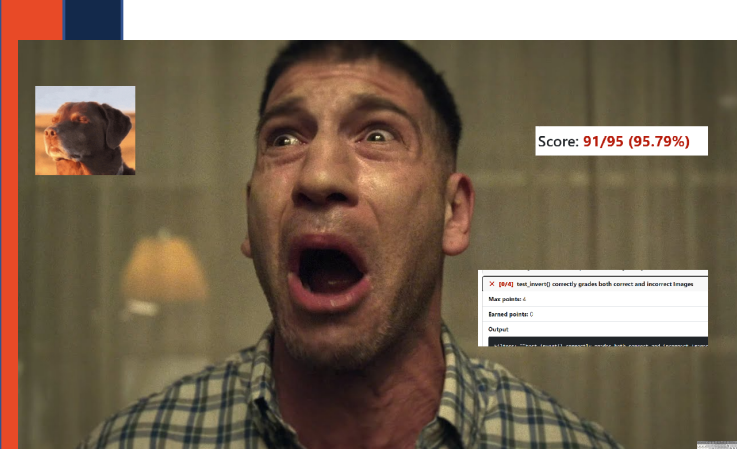
Decompose problem into supporting data structures

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!