Static / Non-static vars + methods

Static variable = **CLASS** variable
- shared between ALL instances
- changes are shared
- can be called from instance / class

non-static = **INSTANCE** variable
- private to each instance
- changes NOT shared
- callable from an instance

* if nested class = static cannot access parent vars

STATIC method
- universal (don't need instance)
- cannot modify instance vars OR call non-static methods → comp error

non-static methods
- specific to instance ↗ call w/ instance
- modify both instance AND class
- run static & non-static methods

Copying

* Primitives are COPIED
* References have POINTERS
  ↳ changes to same reference will impact both pointers

* Strings = OBJECTS + IMMUTABLE

```
int a = 2;
int b = a;
int[] alist = {1,2,3};
int[] blist = alist;
```

a | 2
b | 2
alist → [1,2,3]
blist → [1,2,3]

Datatypes

Arrays = fixed
length → size

Map<String, String> map = new HashMap<>();
Set<Integer> set = new HashSet<>();
int[] x = new int[3];
int[] y = {1,2,3};

lists (Resizable) : List<String> = new ArrayList<>();
add, set, size methods

x instanceOf Integer
↑ var ↑ class

Inheritance

- extends : classes **IS-A RELATIONSHIP**
- implements : interfaces

Dog d = new Corgi();
↑ static type ↓ used @ COMPILE TIME
↑ dynamic type ↓ used @ RUNTIME → **OUTPUT**

- **COMPILE ERRORS** ↗ no runtime
  1. when trying to instantiate interface
  2. Type mismatch (ie SoccerPlayer and Athlete)
  3. Using instance variable in static method

Runtime error: infinite loop, out of bounds

* super keyword → access hidden + overriden methods

DMS

① compile time: looks for method signature to lock in
  ↳ check **static** / compile type
② runtime: looks for method matching signature
  ↳ check **dynamic** type
* check type in parameters

```
Music mr = new Rock();
mr.play(i);
mr.play(g);
```

↳ play(inst) signature from Music()

overriding → subclass w/ same signature as superclass
overloading → multiple methods w/ same name, diff parameters

Tests

- assertEquals
- assertThat(x).isEqualTo(y)

Casting

- forces compile-time type of any expression → change static type
- compile error: incompatible static types
- runtime error: cast more specific than dynamic (dynamic is not a")

Lists

- interface → various implementations (dynamic sizes)
- void add(T item);
- T get(int idx);
- void remove(int idx);
- boolean contains(Object o);
- int size();

List<Integer> lst = new ArrayList<>();

* check pointers!

*** Treeset in sorted + ascending**

Sets

: collection of unique elements w/o order

import java.util.Set
import java.util.HashSet;
Set<String> s = new HashSet<>();

- .add()
- .contains()
- .size()

Maps

: set of key/value pairs

put(K key, V value) : put key value pair into map
V get(K key) : get value corresponding to key
size()

comparators / comparable

- any object that needs to be compared : implement compareTo()
- implement Java's Comparable<T> interface
- implement Comparator<T> when using custom compare()

(neg) if o1 < o2 (pos) if o1 > o2 (0) if o1 = o2

```
public class Dog implements Comparable<Dog> {
    public int compareTo(Dog uddaDog)
        return this.size - uddaDog.size;

private static class NameComparator implements Comparator<Dog
    public int compare(Dog a, Dog b)
        return a.name.compareTo(b.name);

public static Comparator<Dog> getNameComparator()
    return new NameComparator();
```

Iterators

* for-each loop

- for something to be iterable it must include method that returns an iterator

*** create list as var ***

```
public interface Iterator<T>
    boolean hasNext();
    T next();

public interface Iterable<T>
    Iterator<T> iterator();
```
↳ return iterator obj

*** call .iterator on list**

① get new iterator object
② check items still left w/ hasNext()
③ next() to return nex

toString() : override to have custom string representation
equals() : override for comparing ref types
== : compares memory addresses usually used for primitive types

Asymptotics

Big O: upperbound : could grow ≤ f(x) — at most as fast

Big Ω: lower bound: could grow ≥ f(x) — at least as slow

Big Θ: TIGHTEST bound : when upper/lower converge to same value

	ordered array	bushy BST	HashTable	Heap
add	Θ(N)	Θ(log N)	Θ(1)	Θ(log N)
getSmallest	Θ(1)	Θ(log N)	Θ(N)	Θ(1)
removeSmallest	Θ(N)	Θ(log N)	Θ(N)	Θ(log N)

arithmetic:

$1+2+3+\dots+n \in \Theta(n^2)$

$1+3+5+\dots+\log(n) \in (\log^2 n)$

$1+\log(1)+\log(2)+\dots+\log(n) \in \Theta(N\log N)$

geometric:

$1+2+4+8+\dots+n \in \Theta(n)$

$1+3+9+27+\dots+n\log n \in \Theta(n\log n)$

Best vs Worst case: represented w/ tight bound Θ

- Best: exit as fast as possible
- Worst: exit as slow as possible

$O(1) < O(\log N) < O(N) < O(N\log N) < O(n^2) < O(x^n) < O(n!)$

WRITE OUT WORK FOR DIFF VALUES N

```
for (i=0; i<n; i++) {//code}  → power of n
for(i=1; i<n; i*=2) {//code}  ⎤
for(i=n; i>1; i/=2) {//code}  ⎦ → factor of logn
```

Recursive calls

1. Runtime of single layer
2. Draw tree based on # of calls
3. sum work / layer
4. sum up layers

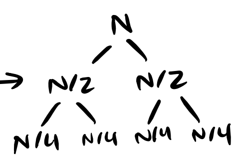
*look at height of tree

$1+2+\dots+n = O(n^2)$

$1+2+4+8+\dots+n = O(n)$

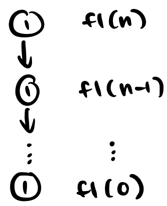
$1+2+3+\dots+\log n = (\log N)^2$ ← largest term²

```
public f1 (int N)
  if N==0
    return
  f1(N/2)
  f1(N/2)
```



log N layers, N work / layer → $O(N\log N)$

```
public f1(int n)
  if n<=0
    return 0
  else
    return n+f1(n-1)
```

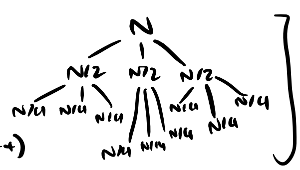


*BREAK UP SUMS

1 work * N levels = N

$S_n = \sum_{i=1}^{N} a_i = \frac{n(a_1+a_n)}{2}$

```
public f1 (int N) {
  if (N==1)
    return
  for(int i=0; i<=N; i++)
    print(N)
  f1(N/2)
  f1(N/2)
  f1(N/2)
```



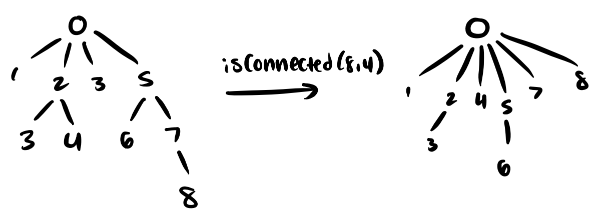
log N layers, $\frac{3}{2}$ layer · N work

*last layer dominates

$O(n^{\log_b a})$

$\sum_{i=1}^{\log N} 2^i = 1+2+4+8\dots+n = 2^{\log n+1}-1 = O(n)$

Path Compression : tie all traversed nodes to root (when isConnected() is called)



Disjoint Sets

```
public interface DisjointSet
  void connect(x,y)          ← connect nodes x & y
  boolean isConnected (x,y)  ← true if x & y connected
```

* means on avg

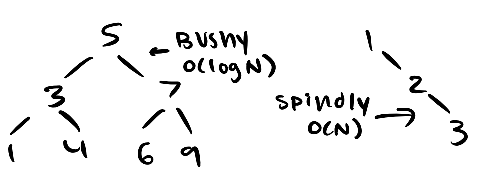
	constructor	connect()	isConnected()
QuickFind = array of integers	Θ(N)	O(N)	O(1)
QuickUnion = stores parent of each node & merges by changing parents	Θ(N)	O(N)	O(N)
WQU = same as QU, merges smaller into larger (reduce stringiness)	Θ(N)	O(log N)	O(log N)
WQU w/ path comp: set parent of node to set's root whenever isConnected(a,b) is called	Θ(N)	O(log* N) Θ(1*) long run	O(log* N) Θ(1)* long run

Stacks : LIFO
Last in, first out

Queues : FIFO
First in, first out

BSTs

1. Node serves as root for smaller tree
2. Node in left subtree < root
3. Node in right subtree > root



Insert: start from root : < root → move left, > root → move right

create new node when we hit null

*order of insertion → height

To traverse a tree
- use nodes and use left and right pointers to move down tree

Delete: no children → remove node

1 child → child replaces deleted node (recurse until leaf),

2 child → replace w/ leftmost node on right or rightmost on left

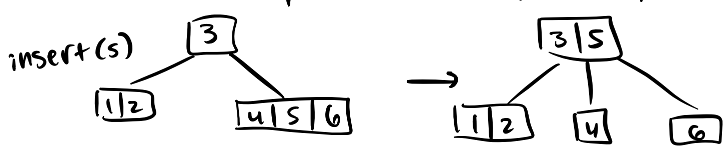
BTrees / 2-3 trees

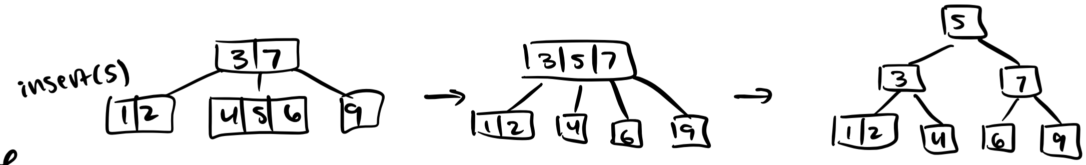
*BALANCED !!

*minimize split & pop for minimum tree

- each node up to 2 items & 3 children
- insert INTO existing node
- < all value → left
  > all value → right
  in between → middle

*if too many node items: split and push up to parent



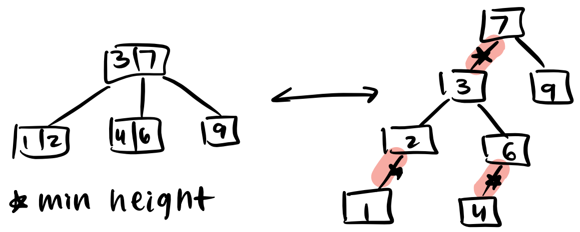


Θ logN to find node
*lower worst case runtime

LLRB

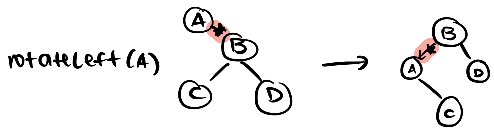
- same structure: use red-links to rep nodes w/ multiple values

*rotateRight → cascading w/ colorflip
*root = black → never colorflip root
*rotateLeft w/o colorflip

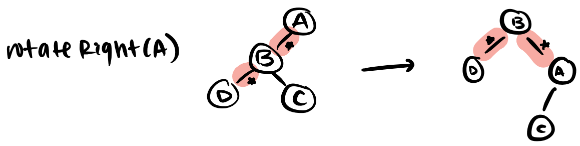


*min height = binary tree
*must have same # of black links from root to null nodes

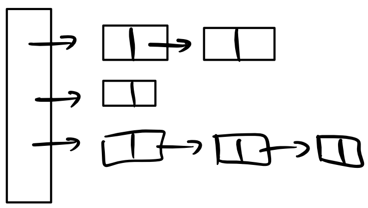
insert w/ redlink → apply fixups

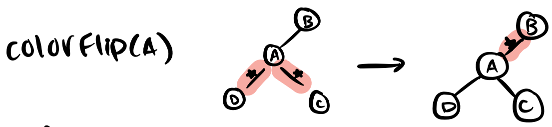


leftleaning violation (NO right leaning red links)



4 node violation (NO consecutive left leaning red-links)





temp 4 node violation (no 2 red children)

Hashing

data → hashcode → Math.floorMod(HashCode, capacity)
→ index to buckets

Hash function: map object w/ integer
① Deterministic H(x) = H(x) → same value for any x
② Equal for value that's .equals() H(x) = H(z) = y if x.equals(z)

GOOD IF ① uniformly distributed
② Fast to compute
** .equals() matches comparing hashcodes **

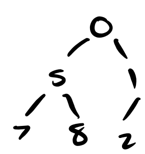
Insertion: ① compute hash key - obj.getHashCode();
② Find bucket: H(key) % arr.length
③ Scan nodes in bucket: if key exists → update for HashMap, nothing for HashSet
if not → insert end of list

Resize after load factor reached
Load factor = N/M (N ↑ # items, M ↑ # buckets)

amortized: O(1) for search, insertion, deletion
worst: O(N) → lots in same bucket
resize: O(N) → O(1) inserted N times

Heaps

: represented as arrays
① root stored @ index 1 (not 0)
② left child @ index 2i
③ right child @ index 2i+1



*bubbling up = linear
*min-heap: every node can be ≤ children
→ [-, 0, 5, 1, 7, 8, 2]

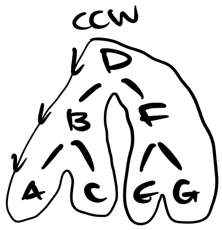
	Best	Worst
insert	Θ(1)	Θ(logN)
find Min	Θ(1)	Θ(1)
remove Min	Θ(1)	Θ(logN)

*complete tree: every level full (except last) and all nodes are far left as possible

Insertion: insert into next available → bubble up
deletion: swap bottom rightmost w/ root → sink down
getSmallest: return root

Priority Queue: like queue where elements sorted on priority (ie min/max)

Graphs



DFS: visit each subtree recursively (w/ stack)
adjacency lists: nodes connected to each node
fringe = datastructure to keep track of nodes to visit
root = where we start traversing

DBFACEG → * level ordering = BFS

BFS: level based w/ a queue
items that are 1 edge
2 edge
. .

BFS(G): $\Theta(V+E)$
Add G.root to fringe
while fringe not empty
pop node from front → process
for each IMMEDIATE neighbor
add child to fringe

TREE TRAVERSAL

① Preorder: * visit crossing LEFT
print(x.key)
preorder(x.left)
preorder(x.right)
mark parent then its child
(visit, go left, go right)
DBACFEG

② Postorder * cross RIGHT
postorder(x.left)
postorder(x.right)
print(x.key)
mark all children then parent
ACBEGFD

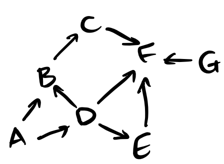
③ In-order : cross BOTTOM
inorder(x.left)
print(x.key)
inorder(x.right)
mark left → self → right
ABCDEFG

General Graph Traversals

BFS: in order of distance
Pre-order: visit, go to children → process node as soon as it enters stack myself, then all children
Post-order: go to children, visit → process node as soon as it leaves all children, then myself
In-order: N/A

* For BFS: distance to all items on queue is always k or k+1

DFS:
initialize fringe (empty stack)
while fringe not empty:
pop vertex off fringe
if vertex not marked:
mark + visit vertex
for each neighbor of verte
if neighbor not marked
push to fringe

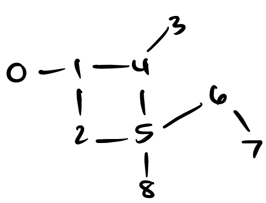


DFS pre-order: A B C F D E G

DFS post-order: F C B E D A G

① Go A → B → C → F & return F → C → B
② Go A → D → E & return E → D → A
③ Go G & return G

*



DFS preorder : 012543678
postorder : 347685210
* figure out how to break ties & remain consistent

① Go 0 → 1 → 2 → 5 → 4 → 3
return 3 → 4
② Go 5 → 6 → 7 & return 7 → 6
③ Go 5 → 8 & return 8
④ return 5 → 2 → 1 → 0

	Access	Search	Insertion	Deletion
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Doubly LL	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
BST	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
LLRB	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Heap	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

$T(n) = aT(\frac{n}{b}) + O(n^d)$

$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^{\log_b a}) & d < \log_b a \\ O(n^d \log n) & d = \log_b a \end{cases}$

a = # of func being called recursively
b = # of work being divided
d : exponent of work done on each level

Sorting

	Memory	runtime	Notes	Stable (if order of = items preserved)
Heapsort	Θ(1)	Best: Θ(N) Worst: Θ(NlogN)	Bad caching	NO ✗
Insertion	Θ(1)	Best: Θ(N) Worst: Θ(NlogN)	Θ(N) if almost sorted	Yes ✓
Merge	Θ(N)	Θ(NlogN)		Yes ✓
Quicksort	Θ(log N)	Best: Θ(NlogN) Worst: Θ(N²)	fastest compare sort	No (typically)
Counting	Θ(N+R)	Θ(N+R)	alphabet keys only	yes ✓
LSD	Θ(N+R)	Θ(WN+WR)	strs of alphabet keys only	Yes ✓
MSD	Θ(N+WR)	Best: Θ(N+R) Worst: Θ(WN+WR)	bad caching	Yes ✓
Selection	Θ(1)	Θ(N²)		NO ✗

N: # of keys W: width of longest key

R: size of alphabet *: constant compare time

comparison sorts: at least NlogN

counting sort: create new arr & copy item w/ key i into ith index

Selection: swap minimum from unsorted to the front
* front items sorted first

7 5 4 2
2 5 4 7
2 4 5 7

Insertion: *sorted & unsorted half
look @ new item & insert in correct spot on sorted half

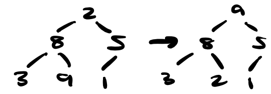
[2 8] 5 3 9 4
[2 5 8] 3 9 4
[2 3 5 8] 9 4
[2 3 5 8 9] 4
[2 3 4 5 8 9]

Best case: alr sorted

Heap: sort into max heap and keep selecting max/top element to place into sorted partition @end

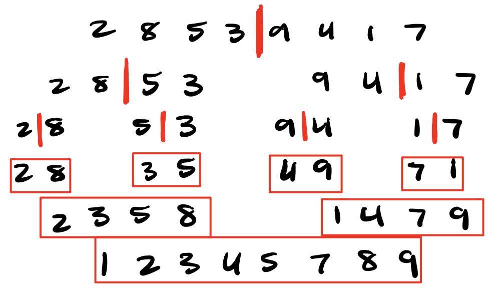
Best when all elem =

2 8 5 3 9 1
9 8 5 3 2 1 heapify
1 8 5 3 2 9 "delete" 9



Merge: *splitting in half & recombine → divide & conquer

Divide into equal parts, recursively sort halves, merge results



Hoare ex

(17) 15 19 32 2 26 41 17 17 — 3 way merge
17 15 17 32 2 26 41 17 19
17 15 17 17 2 26 41 32 19
2 15 17 17 17 26 41 32 19

RESET!

Quick Sort: *using pivot
- choose pivot
- everything lower ← left
- everything higher → right

left = 1st pivot
- L pointer dislikes ≥
- G pointer dislikes ≤
- Dislike + stop → swap
- L overcome G: swap pivot w/ G pointer & repeat

Hoare in-place partitioning

* random pivots
* shuffle before sorting
* Best pivot is median

(18) 7 22 34 99 18 11 4
(7) 11 4 (18) 18 22 34 99
4 (7) 11 18 18 22 34 99

Best = pivot lands in middle Θ(NlogN)

Worst = pivot @ beginning Θ(N²)

requires stable subroutine

LSD: sort each digit indep from rightmost digit to left

582 675 591 189 900 770
900 770 591 582 675 189
900 770 675 582 189 591
189 582 591 675 770 900

*look at groups of sorted digits

Best case: Θ(N+R) w/ only 1 pass of top digit

Worst: Θ(WN+WR) w/ looking @ every char

does NOT require stable subroutine

MSD: Sort each digit from left to right

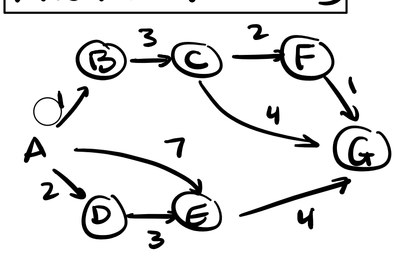
582 675 591 189 900 770
189 582 591 675 770 900
189 582 591 675 770 900

Dijkstra's (SPT): shortest path from one node to every other node in graph

1. Pop node from front of PQ
2. Add/update distances of all children
3. Resort PQ
4. Finalize distance to current node from root
5. Repeat while PQ not empty

use PQ ← heap

NO NEG WEIGHTS



1 Visit A: PQ: B, D, E

	A	B	C	D	E	F	G
dist	0	1	∞	2	7	∞	∞
edge	-	A	-	A	A	-	-

2 Visit B PQ: D, C, E

	A	B	C	D	E	F	G
dist	0	1	4	2	7	∞	∞
edge	-	A	B	A	A	-	-

3 Visit D PQ: C, E

	A	B	C	D	E	F	G
dist	0	1	4	2	5	∞	∞
edge	-	A	B	A	D	-	-

4 Visit C PQ: E, F, G

	A	B	C	D	E	F	G
dist	0	1	4	2	5	6	8
edge	-	A	B	A	D	C	C

5 Visit C PQ: F, G

	A	B	C	D	E	F	G
dist	0	1	4	2	5	6	8
edge	-	A	B	A	D	C	C

6 Visit F PQ: F, G

	A	B	C	D	E	F	G
dist	0	1	4	2	5	6	7
edge	-	A	B	A	D	C	F

★ KEEP TRACK OF TOTAL DIST FROM ROOT

directed ✓ undirected ✓ cyclic ✓

A*: Dijkstra but w/ heuristic (SPT)

use: (distance from start) + (heuristic)

* Admissible: heuristic val NEVER exceeds true distance: heuristic(v, target) ≤ trueDist(v, target)
* Consistent: heuristic(v, target) ≤ dist(v, w) + heuristic(w, target)

SPT: $O((E+V)\log V)$ → $E\log V$ | priority, $V\log V$ | add/remove vertices

Minimum Spanning Tree: minimizes global sum of weights

✱ MULTIPLE ON SAME GRAPH

shortest net path around graph to hit ALL nodes that's NOT cyclic

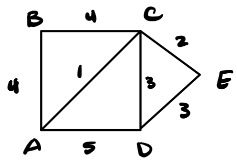
Kruskal's Algorithm

while there are still nodes not in MST:

- Add lightest edge that doesn't create a cycle
- Add endpoints of that edge to set of nodes in MST

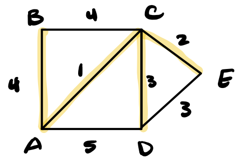
✱ Edges sorted in non-decreasing order of weight

✱ start w/ vertex carrying minimum weight



AC
CE
CD
AB

→



✱ doesn't have to be adjacent edge

✱ uses WQU and path compression

✱ V-1 edges

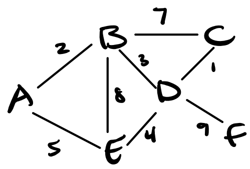
✱ Sorting: $O(E \log E)$

✱ Total: $O(E \log V)$

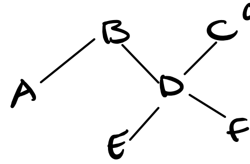
PRIMS ALGORITHM

works w/ neg edge weights

1. start w/ any node
2. Add that node to nodes in MST
3. while there are still nodes NOT in MST:
   - Add the lighest edge from node in MST that leads to unvisited node
   - Add new node to set of MST nodes



AB
BD
DC
DE
DF



cut property: smallest edge spanning current vertex & others always in MST

Time complexity: $O(V^2)$

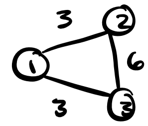
cut property: given any cut, any min weight crossing edge in MST

↑ assignment of graphs nodes to 2 non-empty sets

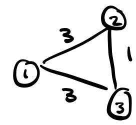
↑ edge that connects node from one set to node from other set

unique edge weights = 1 MST

same



different



Duplicate edge weights → different MSTs w/ diff tiebreaking

Kruskal's	# of times	Time per op	Total time
insert	E	$O(\log E)$	$O(E \log E)$
delete min	$O(E)$	$O(\log E)$	$O(E \log E)$
union	$O(V)$	$O(\log^* V)$	$O(V \log^* V)$
is Connected	$O(E)$	$O(\log^* V)$	$O(E \log^* V)$

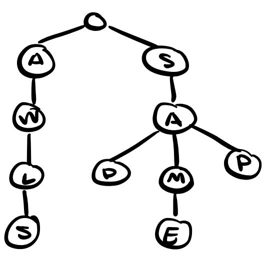
Prim's	# of times	Time per op	Total time
PQ add	V	$O(\log V)$	$O(V \log E)$
PQ delMin	V	$O(\log V)$	$O(V \log E)$
PQ decrease Priority	$O(E)$	$O(\log V)$	$O(E \log V)$

		runtime (E>V)	Fails for neg weights
SPT	Djikstra's	$O(E \log V)$	
MST	Prim's	$O(E \log V)$	= Djikstra's
MST	Kruskal's	$O(E \log E)$	WQUPC
MST	K's sorted	$O(E \log^* V)$	WQUPC
	A*	$O((V+E) \log V)$	

DFS	$O(V+E)$
BFS	$O(V+E)$
Djikstra's	$O((V+E) \log V)$
A*	$O((V+E) \log V)$
Prim's	$O((V+E) \log V)$
Kruskal's	$O(E \log E)$

Tries: each node corresponds to single char

a
awls
sad
sam
same
sap



Insertion: $O(n)$
↑ key length

Prefix search: $\Theta(M)$ len of str

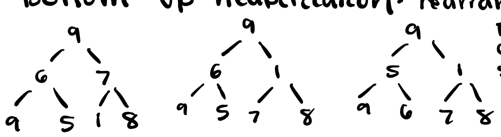
PQ: elements sorted on priority (min/max)

.add()
.size()

to keep K max elems in min heap:

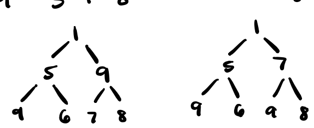
```
for (i<n):
  pq.add(i);
  if (pq.size() > k)
    pq.removeSmallest
```

Bottom-up heapification: treat array as heap, rearrange nodes



BUBBLE DOWN on every node starting from bottom

$\Theta(n)$



Top-down heapification: start w/ empty heap and inserts all elements into it

worst case: $\Theta(n \log n)$