	Storage operation	Retrieval operation	retrieved by
List	add(key) insert(key, index)	get(index)	index
Map	put(key, value)	get(key)	key identity
Set	add(key)	contains(key)	key identity
PQ	add(key)	getSmallest()	key order aka key size
Disjoint Sets	connect(int1, int2)	isConnected(int1, int2)	look int values

	Access	Search/Contains	Insertion	Deletion
Array (resizing)	Θ(1)	Θ(n)	Θ(n)	Θ(n)
Stack/ Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)
Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)
Hash Table	no ordering	Θ(1)/Θ(n) good/bad h.f.	Θ(1)/Θ(n)	Θ(1)/Θ(n)
BST	Θ(log n)/Θ(n) bushy/spindly	Θ(log n)/Θ(n)	Θ(log n)/Θ(n)	Θ(log n)/Θ(n)
B-tree aka B-tree	Θ(log n)	Θ(log n)	Θ(log n)	Θ(log n)
Heap	Θ(1) getMin() costs removeMin() NOT	N/A	Θ(log n) changePriority() is log n	Θ(log n) last elem to top, bubble down
LLRB	Θ(log n)/Θ(n)	Θ(log n)/Θ(n)	Θ(log n)/Θ(n)	Θ(log n)/Θ(n)
Trie	No ordering	Θ(L)	Θ(L)	Θ(L)

	Best case	Worst case	Additional Space	Notes
Selection Sort	Θ(n²)	Θ(n²)	Θ(1)	not stable
Merge Sort	Θ(n log n)	Θ(n log n)	Θ(n)	Stable (better than quicksort for LL)
Insertion Sort	Θ(n)	Θ(n²)	Θ(1)	Stable
Quick Sort	Θ(n log n)	Θ(n²)	Θ(n)	dep on strat.
Heap Sort	Θ(n)	Θ(n log n) ~ mergesort	Θ(1)	not stable

Some Sums

$1+2+3+\dots+Q = Θ(Q(Q+1)/2 = Θ(Q^2)$

$1^k+2^k+\dots+Q^k \quad (k \geq 0) = Θ(Q^{k+1})$

$1+q+q^2+\dots+Q^Q = 2(2^Q)-1 = Θ(2^Q)$

$K^0+K^1+\dots+K^Q \quad (K>1) = Θ(K^Q)$

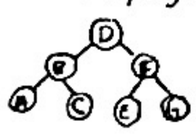
```
for(int i=1, sum=N; i+=sum){
  for(int j=0; j<i; j++){
    ... }}
```

$1+2+4+\dots+N \leq Θ(N)$ N=k^a

$k^0+k^1+\dots+N = Θ(N)$

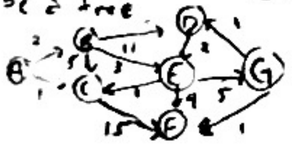
Traversals – Tree

Preorder: current, left, right
Inorder: left, current, right
Postorder: left, right, current.



Pre: DBACFEG
In: ABCDEFG
Post: ACBEGFD
LEVEL order: DBFACEG

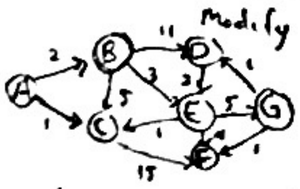
Shortest paths –
The shortest will always be a tree



Dijkstra's Algo – Best first search

1. Add all vertices to fringe PQ
2. Remove closest vertex from PQ, "relax" all edges pointing from vertex.
   ↳ if vertex gives better dist., add the edge, update the vertex in the fringe.

All edges start w/ distTo = ∞, slowly modify and change EdgeTo



Node	distTo	edgeTo
A	0	–
B	2	A
C	1	A
D	13→11	B→G
E	5	B
F	16→9	C→E
G	10	

Arrows indicate change.

Guaranteed to be optimal as long as no neg. edges!

edge	Step
A	1: A=0, B=2, C=1
C	2: F=16
B	3: D=13, E=5
E	4: F=9, G=10
F	4.5: nothing!
G	5: D=11
D	6: nothing!

fringe per step

1: [(C,1), (B,2), (D,∞), ...]
2: [(B,2), (F,16), (D,∞) ...]
3: [(E,5), (D,13), (F,16), (G,∞)]
4: [(F,9), (G,10), (D,13)]
5: [(D,11)]
6: []

overall eff: O(E log V)

- add O(V log V)
- removeSmallest O(V log V) → V ops at ons, each log V time
- changePriority O(E log V) → E operations each log V time

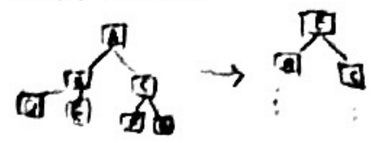
Disjoint Sets

Linked Sets

QuickFind

WeightedQU

Hibbard deletion



if A deleted, choose rightmost from left(E) or leftmost from right(F)

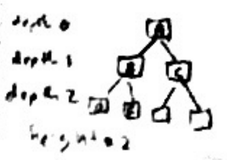
BST

```
BST find(BST T, Key sk){
  if(T==null) {return null;}
  if(sk equals(T key)){return T;}
  else if(sk > T.key){return find(T right, sk)}
  else if(sk < T.key){return find(T left, sk)}
}
```

```
BST insert(BST T, key ik){
  if(T==null){return new BST(ik)}
  if(ik<T.key){T left = insert(T left, ik)}
  <same for right>
  return T;
}
```

Random trees are bushy!

Height of BST ~ log₂(N)
So search Θ(log N)

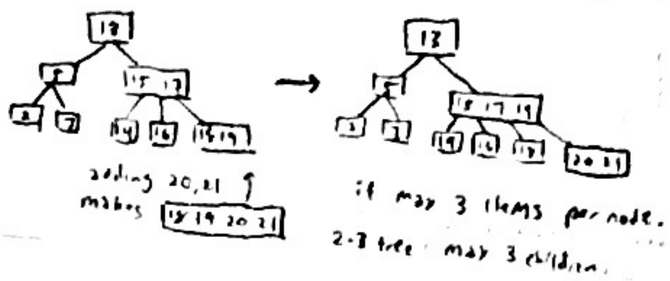


Avg depth and height def. same
Worst case 3 comparisons (h+1)
Avg case = Avg depth of nodes

Worst case Θ(N) height
Best case O(log N) height

B-trees

2 invariants: → Guarantee bushy tree!
- ALL leaves same dist. from the root
- non-leaf nodes with K items must have K+1 children



if max 3 items per node.
2-3 tree: max 3 children.

ALWAYS
move up left-middle
spawning node uses Θ(L) compares, but ok, L const.
Allowed children = # Allowed items + 1

overall height Θ(log N)
overall runtime Θ(log N)

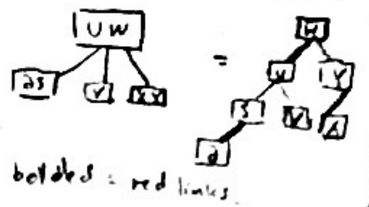
LLRB

invariant: 2-3 tree of height H, corresp LLRB:
height: H + (#red) = 2H+1
(black)

insertion rules:
- if inserting right w/ no siblings, rotate left on parent
- if inserting right w/red node to left, flip the tree so red node becomes parent
- if inserting left w/ red parent, we have double left lean. rotate right on parent, then color flip
- if inserting left on normal parent, no changes needed!

no node has 2 red links (then it be 4-node)

All paths root to null have same # of black links, therefore LLRBs balanced!

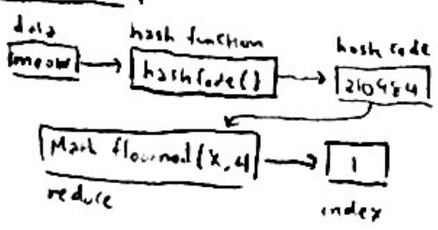


bolded = red links

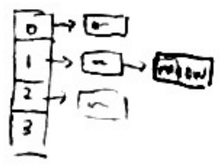
rotateLeft(G): make G the left child of its right node G's right node becomes the left node of its child.

rotateRight(G): same but reversed.

Hash Tables



Linked List.



TO achieve evenly dist hash table, random hashcodes are best.
Limitation: if hashcode for 2 of same object diff, then insert could put dupes in, and contains can't always find the object

floorMod used to give negative hashcodes valid indices

Warnings
1. Mutable objects should never be stored by hash ↳ hashcode changes!
2. If equals overridden, hashcode should also be

Hash codes:
Java uses base 31.
1 Never even (overflow)
2 Small prime.
if M = # of bins, N items; resize when N/M > some constant

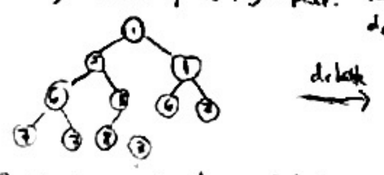
3 properties of valid hashCodes
1. Hashcode returns an integer.
2. Calling hashcode twice on the same unchanged obj should yield same int.
3. 2 equal obj should have the same hashCode

Heaps and PQ

To store N highest value objects.

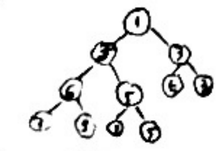
Binary min-heap: each node less than or equal to children
Min-heap ↑ complete: when missing items at bottom level, all nodes as far left as possible

Heap add: add to end of heap temporarily "swim" up to right place.

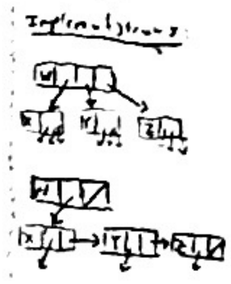


3 swims up to the 2nd layer of 5.

Heap delete: remove top node, replace with item at end of heap. "Swim" down.



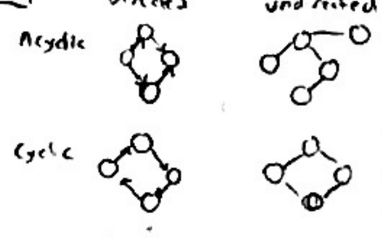
3 round to top, swims down to place.



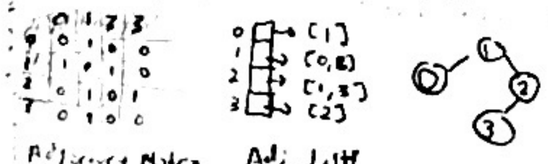
Assume complete, use only key array

Any node that doesn't have half total # of nodes as children/descendants can be median

Graphs



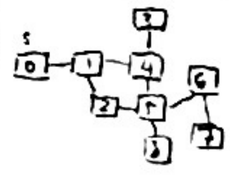
Simple graph: no edges that connect vertex to itself
no two edges connect same vertices
Vertices are the nodes, edges are lines.
Simple path: no repeated vertices path



Adjacency Matrix: space eff for lots of edges. Adding edge low eff.

Adj. List: Space eff for few edges returning list of edges better Θ(V+E) vs Θ(V²)

DFS – depth first search



DFS preorder: 0 2 5 4 3 6 7 8
remove self from fringe, add children
DFS postorder: 3 4 7 6 8 5 2 1 0
don't visit self until visited all children
BFS order: 0 1 2 4 5 3 6 7 8
sort in order of distance from source

	s-t paths	depth	O(V+E) time O(V) space
	s-t shortest paths	breadth	O(V) time

Graph impl.

```
Public class Graph {
  Public Graph(int V) { ... }
  Public void addEdge(int v, int w)
  Iterable<Integer> adj(int v){...}
}
Public void DepthFirstPaths {
  priv bool[] marked;
  priv int[] edgeTo
  priv int s;
  Public DepthFirstPaths(Graph G, int s){... dfs(G,s);}
priv void dfs(Graph G, int v) {
  marked[v] = true;
  for(int w: G.adj[v]){
    if(!marked[w]){
      edgeTo[w] = v;
```

```
Public class BreadthFirstPaths{
  Private bool[] marked,
  Priv int[] edgeTo;
  ...
Private void bfs(Graph G, int s){
  Queue<Integer> fringe = new Queue<Integer>();
  fringe.enqueue(s),
  marked[s] = true,
  while(!fringe.isEmpty()){
    int v = fringe.dequeue()
    for(int w: G.adj(v)){
      if(marked[w]){
        fringe.enqueue(w),
        marked[w] = true,
```

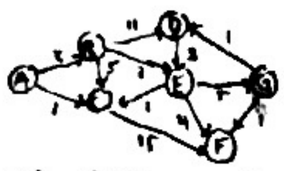
RB Tree impl.

colorFlip(node){}
↳ flips color of node and its children

rotateRight(node){}
rotateLeft(node){}

A* - Dijkstra's with end goal in mind.

Solve all vertices into fringe PQ, order of dist(source, v) + h(v, goal) h is arbitrary, measure of "dist" from goal



Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	2	A	3
C	1	A	15
D	1.63	B	1
E	5	B	1
F	7	E	∞
G	10	E	0

fringe pop

A 1: [(B,5), (C,15), (D,∞), ...]

B 2: [(E,16), (D,15), (C,16), ...]

E 3: [((G,10), (D,15), (C,16), (F,∞)]

next visited is G, so done!

Problem

Find path from s to every reachable vertex	depth-first: O(V+E) time, O(V) space
Find shortest path from s to every reachable vertex	breadth-first: O(V+E) time, O(V) space
Shortest paths considering weights from s to reachable vertices	Dijkstra's: O(E log V) time, O(V) space
Shortest weighted path from s to target vertex	A*: O(V) space

Spanning tree is subgraph of undirected graph that is
- connected, acyclic, includes all vertices

MST is indep. of "source". Cut - partitioning graph nodes into 2 nonempty sets. For any cut, the minimum weight crossing edge is in the MST.

Selection sort

Find smallest in unsorted portion. Move to end of sorted portion. Selection sort remaining unsorted.

32 15 2 17 19 26 41 17 17

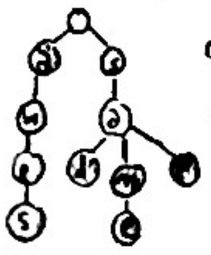
2 32 15 17 19 26 41 17 17

⋮

$\Theta(N^2)$ time if using arr.

$\Theta(1)$ Memory if swap in-place

Trie - alternative to hashMap / BST when all data of 1 type.



Contains "sam" - true, blue node

Contains "d" - true, blue node

Contains "sa" - false, white node.

Search "miss" — end on White node, or "fell off" the tree (e.g. sax).

Add runtime $\Theta(n)$ | contains $\Theta(1)$ or in terms of length of key L: $\Theta(L)$ | $\Theta(L)$

indep of key #!

	key type	get(x)	add(x)
Balanced BST	Comparable	$\Theta(\log N)$	$\Theta(\log N)$
Hashing w/ Hash table	hashable	$\Theta(1)$	$\Theta(1)$
Digital search tr.	chars	$\Theta(1)$	$\Theta(1)$
Trie	strings	$\Theta(1)$	$\Theta(1)$

Main appeal: prefix matching!

longest prefix of ("sample") → "sam"

```
public class TrieSet{
    private static final int R = 128;
    private Node root;
    private static class Node{
    private boolean isKey;
    private DataIndexedCharMap(Node) next;
    Node(boolean b, int R){
        isKey = b;
        next = new DataIndexedCharMap<>(R);
    }
}
public class DataIndexedCharMap<V>{
    private V[] items;
    public DataIndexedCharMap(int R){
        items = (V[]) new Object[R];
    }
```

collect() → empty list of words x.
For char c in root.next.keys(), call colHelp(c, x, root.next.get(c)), return x.
colHelp(Prefix s, List<String> x, Node n)
if n.isKey, then x.add(s). For char c in n.next.keys(), call colHelp(s+c, x, n.next.get(c))

Median identif. Algo - Quick select

- Partition, pivot lands at 2 indices. Next, sort from the bigger side (must contain median!) Repeat until pivot at size/2.

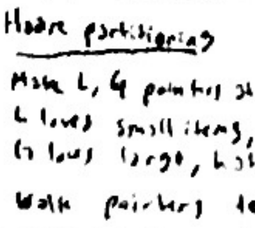
Worst case: $\Theta(N^2)$, on Avg: $\Theta(N)$

Merge Sort

basically splits input into halves of sort those then merges lists.

Merge runtime: $\Theta(N)$, sorting w/ sel sort of each: $\Theta(N^2)$, but with half items.

Overall eff. $\Theta(N \log N)$ both best/worst



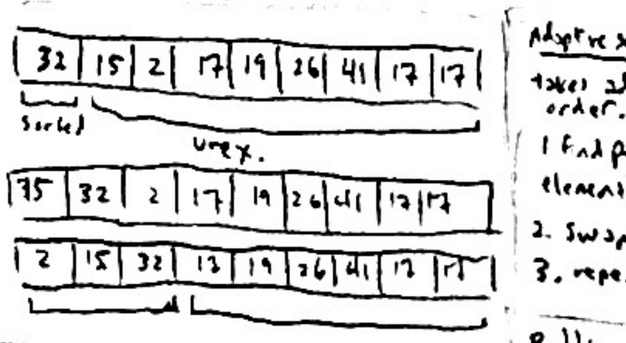
space: $\Theta(N)$

Insertion Sort

Generally: start w/ empty output. Add each item from input to output inserted in the right place.

on arrays w/ small # of inversions, insertion sort is very fast! → $\Theta(N+k)$ where k is # of inversions. On Avg, $k = N^2$, but almost sorted lists, such as $k < cN$ for some defined small c, insertion sort very good!

in-place: repeat for i:0 → N-1 item i is traveling item. swap backwards, until at the right place.



For small lists (N < 15ish) insertion sort the fastest.

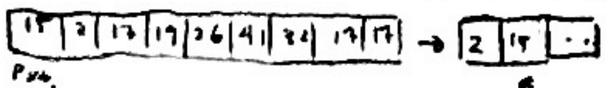
Best case: $\Theta(N)$
Worst: $\Theta(N^2)$
Space: $\Theta(1)$

inversion: pair of elements out of order w/ resp to <. etc.

Quicksort

Partition left most item. Quicksort left half, right.

Partition: item chosen. All smaller go to left, All bigger go to right.



runtime dep. on pivots. Partitioning is $\Theta(K)$, where k = # of items being part.

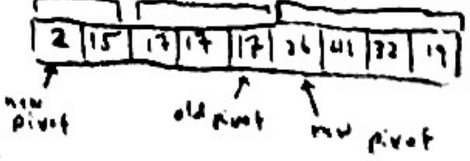
sorted/close to sorted are worst case! Bad elem. = Array of all duplicates.

total work at each level × # of levels

Best case: partition in middle → $\Theta(N \log N)$
worst case: partition in bdsing → $\Theta(N^2)$
On Avg, it's $\Theta(N \log N)$.

Memory: $\Theta(\log N)$

Avoid:
1. random pivot/shuffle
2. approx median input
3. switch sorts if recursion too deep.
4. preprocess to see if quicksort slow

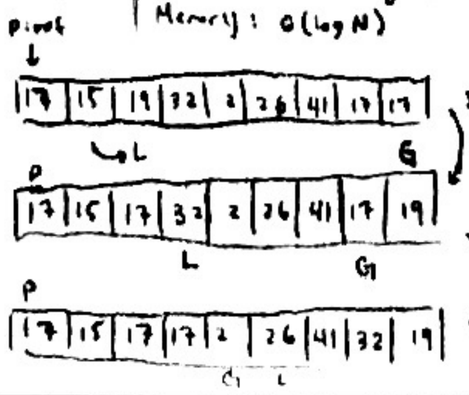


Hoare partitioning

Make L, G pointers at left/right ends.
L loves small items, hates large/equal
G loves large, hates small/equal

Walk pointers to each other, stopping on hated item. When both stop, swap the items L and G are on, and then move them again. Finish when pointers cross.

Swap P with G.



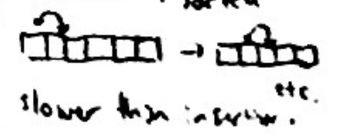
Adaptive sorting

takes advantage of existing order.

1. Find pair of adjacent elements in wrong order
2. Swap them
3. repeat 1 and 2

Bubble Sort:

iterate through list, look at each pair. If out of order, swap them repeat until sorted



etc.

slower than insertion.

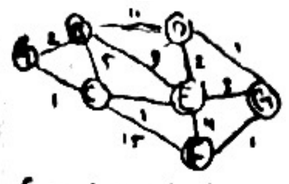
Bogo

while list not sorted, shuffle list.

Best: O(N) in list, sorted

Avg: $\Theta(n \cdot n!)$

Prim's Algo (MST)



Conceptual (slow):
Start from arbitrary start node
- repeatedly add shortest edge connected to node inside MST

	edgeTo	order added
A	-	
B	A	6
C	A	1
D	E	3
E	C	2
F	G	5
G	D	4

	eff.	dequeue	total
PQ add	V		O(VlogV)
PQ delMin	V		O(VlogV)
PQ decrease priority	O(E)		O(ElogV)

Realistic Prim's

insert all vertices into fringe PQ, storing dist, from the order. Remove closest v, relax edges

	distTo	edgeTo
A	0 → -	-
B	2 → -	A
C	1 → -	A
D	2 → -	E
E	1 → -	C
F	15 → 9 → 1 → -	E → G
G	3 → 1 → -	E → D

A 1: [(C,1), (B,2), (D,∞), ...]
C 2: [(E,1), (B,2), (F,15), (D,∞), ...]
E 3: [(B,2), (D,2), (G,3), (F,9)]
B 4: [(D,2), (G,3), (F,9)]
D 5: [(G,1), (F,4)]
G 6: [(F,1)]

→ overall E log V, same as Dijkstra's.

Kruskal's algo

Consider edges in order of incr. weight. Add edge unless cycle is created. Repeat until V-1 edges

	cost	MST
A-C	1	✓ [A-C]
C-E	1	✓ [A-C, C-E]
D-G	1	✓ [A-C, C-E, D-G]
F-G	1	✓ [A-C, ..., F-G]
A-B	2	✓ [A-C, ..., A-B]
E-B	3	✗ [A-C, ..., A-B]
D-E	3	✓ [A-C, C-E, D-G, F-G, A-B, D-E]
G-E	3	
E-F	4	
B-C	5	
B-D	11	
C-F	15	

Done! (created V-1 edges)

Cost on: insert, delMin, union, stream
eff(alg) O(ElogE) O(ElogV) VlogV ElogV

(with heapSort, insert → O(E) bottom-up heapification.)

opt. Kruskal's

insert all edges to PQ. Remove smallest weight edge. Add to MST if no cycle created.

WQU	MST	check cycle with
[A-C]	[A-C]	is (connected (new edge))
[A-C-E]	[A-C, C-E]	if yes, do nothing
[A-C-E, D-G]	[A-C, C-E, D-G]	if no, add to
[A-C-E, D-G-F]	[A-C, ..., F-G]	MST, change WQU
[A-C-E-B, D-G-F]	[A-C, ..., A-B]	
[A-C-E-B-D-G-F]	[A-C, ..., D-E]	

done!

Kruskal's	Kruskal's w/ pre-sorted edges
O(E log E)	O(E log V)

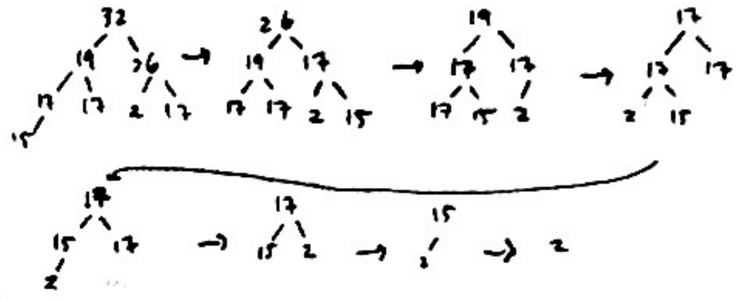
HeapSort

insert all items into max heap. Delete, input arr, make output. Repeat N times:
1 Delete largest item from max-heap.
2 put largest item at end of unused part of output arr.

Heap: | 0 | 41 | 19 | 32 | 17 | 19 | 2 | 26 | 17 | 17 |

size: 9.

essentially pop top of max-heap, then it will reconfigure itself using its swim-up. repeat.



runtime: $O(N \log N) + \Theta(N) + O(N \log N) = O(N \log N)$

items into heap | largest item $\Theta(1)$, N times | removing largest item $O(\log N)$

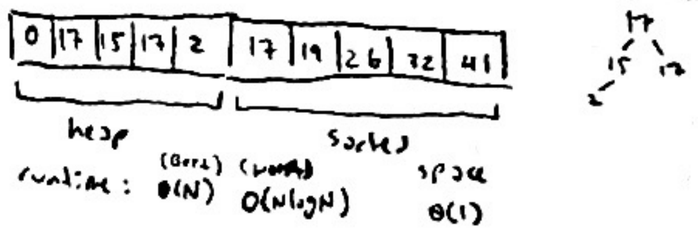
Memory: $\Theta(N)$. To fix, use in-place Heapsort

in-place Heap

1 Convert input to heap
2 Repeat N times: delete largest from max-heap, move deleted item to vacated arr. slot.

Max heap, use bottom-up heapification

Sink nodes in reverse level order.



	(Best)	(Worst)	space
runtime:	$\Theta(N)$	$O(N \log N)$	$\Theta(1)$

Insertion Sort

Generally: start w/ empty output. Add each item from input to output inserted in the right place.

on arrays w/ small # of inversions, insertion sort is very fast! → $\Theta(N+k)$ where k is # of inversions. On Avg, $k = N^2$, but almost sorted lists, such as $k < cN$ for some defined small c, insertion sort very good!

in-place: repeat for i:0 → N-1 item i is traveling item. swap backwards, until at the right place.

For small lists (N < 15ish) insertion sort the fastest.

Best case: $\Theta(N)$
Worst: $\Theta(N^2)$
Space: $\Theta(1)$

inversion: pair of elements out of order w/ resp to <. etc.

Quicksort

Partition left most item. Quicksort left half, right.

Partition: item chosen. All smaller go to left, All bigger go to right.

runtime dep. on pivots. Partitioning is $\Theta(K)$, where k = # of items being part.

sorted/close to sorted are worst case! Bad elem. = Array of all duplicates.

total work at each level × # of levels

Best case: partition in middle → $\Theta(N \log N)$
worst case: partition in bdsing → $\Theta(N^2)$
On Avg, it's $\Theta(N \log N)$.

Memory: $\Theta(\log N)$

Avoid:
1. random pivot/shuffle
2. approx median input
3. switch sorts if recursion too deep.
4. preprocess to see if quicksort slow

Hoare partitioning

Make L, G pointers at left/right ends.
L loves small items, hates large/equal
G loves large, hates small/equal

Walk pointers to each other, stopping on hated item. When both stop, swap the items L and G are on, and then move them again. Finish when pointers cross.

Swap P with G.

Adaptive sorting

takes advantage of existing order.

1. Find pair of adjacent elements in wrong order
2. Swap them
3. repeat 1 and 2

Bubble Sort:

iterate through list, look at each pair. If out of order, swap them repeat until sorted

etc.

slower than insertion.

Bogo

while list not sorted, shuffle list.

Best: O(N) in list, sorted

Avg: $\Theta(n \cdot n!)$