Sums

$1 + 2 + 3 + \ldots + Q = \frac{Q(Q+1)}{2} = \Theta(Q^2)$

$1 + 2 + 4 \ldots + Q = 2Q - 1 = \Theta(Q)$

$1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{Q} = \ln Q = \Theta(\log Q)$

B/W

Access Control:

Private: Only code from same clars can acces

Package private: Vars declared without modifier are declare pack-priv. Other classes in same package can acces but subclasses cannot

protected: classes in shared package and subclasses can access

public: any clars can access

All classes haves: toString, hashcode, equals

equals

① Reflexive x.equals(x)

② Symmetric x.equals(y) = y.equals(x)

③ Transitive: x=y y=z x=z

equals takes an Object param, returns false for null

Iterators / Iterable:

IteratableT> inface: Iterator<T> iterator()

IteratorT>: hasNext(), next()

any class implementing iteratable can be used in an enhanced for-each loop

nested, non static clarses can be instantiated by var.new ClarsName(...);

Comparable<T>: provides int CompareTo(T o)

Comparator<T>: gives int compareTo(O_1, O_2)

Disjoint Sets: Sets where elements can only exist in one set at a time

[1,2] [3] [4] Valid D.S.

quion(a,b) – connects two sets

find(a,b) – are a and b connected

Quick Find: uses array

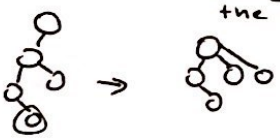
Quick Union: Tree representation to speed union

weighted quick union:

① union smaller tree to larger one

(max dep: log N)

Path compression: Every find call moves a node to the root including terminator



Stacks: LIFO

push: item goes to top of stack

pop: removes top item

Queue: FIFO

add: adds to end of queue

peek: shows item next

poll: removes/returns next item

Heapify: Top-down recursively built

Comparison Based Sort:

pairwise comparison

insertion sort

Selection sort

merge sort

quick sort

Total Order: det. if A is <, >, = to B

Stable: Consistent sort

Order of Growth:

Const	1
log	log N
linear	N
linearithmic	N log N
quad	N^2
cubic	N^3
expon	2^N

Runtime	BST	RBTree
Contain	$\Theta(1)/\Theta(N)$	$\Theta(1)/\log N$
insert	$\Theta(1)/\Theta(N)$	$\log N / \log N$
rem	$\Theta(1)/\Theta(N)$	$1 / \log N$

Streams:

Stream().filter.map.forEach.count

Lambda: (a,b,...) → operations (return)

forEach(func) → System.out::println

Hash Table:

- can provide Θ(1) removal, insert, search
- map (K,V)
- external chaining: uses another d.s. to store elements that are in the same hash table
- Load Factor: size/length => resize len when ratio 0.75
- hashcodes are limited to 2^{32} unique codes
- your buckets are filled by doing item % length

Heaps:

- like trees
- every element has value
- min/max value is the root

Runtime	Avg.	Worst
space	N	N
search	N	N
insert	1	log N
delete	log N	log N
peek	1	1

Binary Heaps (restricted to two children) which act like binary trees with 2 extra invariants

① Completeness: all positions are filled to the left (no holes)

② Heap Property: value of element E must be smaller larger than all of its children

min-heap: root is min-value

Heap Sort: Θ(N log N)

Trees as arrays:

① Root at position 1 (nothing @ 0)

② Left Child of Node @ N is 2N

③ Right Child of Node @ N is 2N+1

④ Parent of node @ N is ½·N

Operations

Insertion

① insert the element into the left most slot

② swim element up

Deletion

① Remove the min/max root

② Put last element as root

③ Sink it

Sorting Runtime:

Sort	Best	Worst	Stable	Note
Insertion	N	N^2	yes	
Tree	N log N	N log N	yes	unbal. worst (N^2)
Selection	N^2	N^2	no	can be stable
Heap	N log N	N log N	no	equal Θ(N)
merge	N log N	N log N	yes	
Quick	N log N	N^2	maybe	three-way stable

Priority Queue:

new PriorityQueue(O1, O2) → O1.compareTo(O2)) min-heap

new PriorityQueue(O2, O2) → O2.compareTo(O1)) max-heap

Streams

- collect(Collector c) reduce(StreamSortUtil::merge) → call most collect
  - Collectors.toList()
- forEach(Consumer)
  - Consumes the stream - terminates

Dijkstra's Runtime: $(|E| + |V| \log |V|)$

Adding one to every edge won't affect Prim's algorithm but will affect Dijkstras

SPT and MST will often times differ - run through Dijkstras to verify

Hoare Partitioning:

1. select pivot
2. go from start and end and find inversion
3. swap, continue until returning final index

pivot selection:

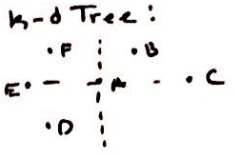
1. Leftmost → worst case behavior
2. median element

DFS: (Strongly connected directed graph) $\Theta(|V| + |E|)$

Dijkstra: $\Theta(|E| \log V)$

Kruskal WUG: $\Theta(|E| \log V)$

k-d Tree:



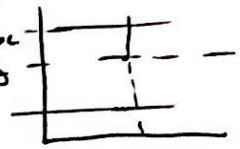
X, Y, X, Y

you will put objects

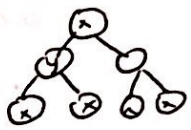
< A to left

> A to right

hypercube partioning



you partion the space



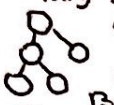
worst

LSD mergesort $\Theta(WN \log N)$

LSD quicksort $\Theta(WN^2)$

- merge: stable
- quick: unstable

Binary Search Trees:



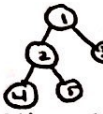
- Each node has left + right child
- height = level (single node = 1)

Balance:
1. no nodes / single is balanced
2. height of left sub tree/right equal
3. L/R tree balanced

Search Invariant:
1. items left of root < key
2. items right of root > key

Traversals

DFS has three orderings*
Pre-Order: Root → Left Tree → Right Tree
Post-Order: Left → Right → Root
In-Order: Left → Root → Right



Pre: 1 2 4 5 3
Post: 4 5 2 1 3
In: 4 2 5 1 3

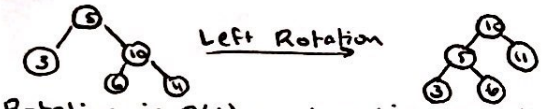
Deletion Algorithm (BST): Delete removal node and replace with in-order successor

	Runtime: Avg	Worst
Space	O(n)	O(n)
Search	O(log N)	O(N)
Insert	O(log N)	O(N)
Delete	O(log N)	O(N)

*Worst-case arises from spindy, unbal. trees

Rotation

1. select root and pivot
2. promote pivot to root and connect subtree to root



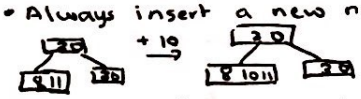
Rotation is O(1) constant time operation
In-Order Invariant: Rotation is correctly performed if in-order traversal is retained

2-3-4 Tree:

- special BTree that guarantees balance

Rules:
1. Node may have 2,3,4 children
2. You must have one more child than key (limit 3 keys)
3. Follows ordering invariant of BST

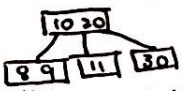
Insertion:
- Always insert a new node in a leaf



- if you try to add a node like 9, you will surpass the key limit so you must do the following

1. Promote the middle node to parent (if root, new node)
2. Split remaining values into two nodes
3. Add normally

* height of 2-3-4 tree will only increase when root is full



2-3-4: answer (log N)Θ on insert, delet, search

Red-Black-Tree (Self Balancing Binary Search Tree)

1. Every node has a color, red/black
2. root is black
3. every red node has black parent/children (no adjacent red nodes)
4. Every path from root → Null (termination) has same number of red/black

Insertion:
1. insert as leaf as red
2. Restore RB property

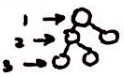
Restore
1. Recolor
2. Rotate

isometry of 2-3-4 trees

Deletion:
1. in-order successor
2. Restore RB properties

Breadth-First-Search (Level Order)

1. Explore nodes on same level
2. Next level



Implemented best in queue like structure
Θ(|V| + |E|) worst-case

Depth-First-Search:

Best represented by a stack

Topological Sort: Only valid on DAGs and also called linearization

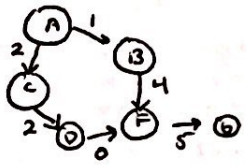
1. Nodes are added to list when indegree=0
2. Remove nodes and repeat

Pre: 6A 61B 169 170 188 61C
Post: 169 170 188 61B 61C 61A
BFS: 61A 61B 61C 169 170 188
Topo: 61A 70 61B 61C 169 170 188

* you always visit nodes in order (alphabetically or numerically)

Dijkstra's + A*: Shortest-Path Algorithm:

- implemented using binary min-pq where priorities are determined by distance traveled + distance to node



Θ(|E| + |V| log|V|)

A*: implemented by adding a heuristic of some kind

1. pq: A 0
2. A
   pq: B 1
   C 2
3. A B
   pq: C 2
   F 5 (B+F weights)
4. A B C
   F 5, D 4
5. A B C D
   pq F 4 → update cost of F to reflect better path

Dij: A B C D F G

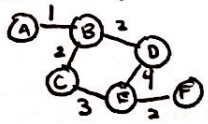
Minimum Spanning Trees:

in a graph of N nodes, there exists a MST that spans all nodes with an edge count of N-1 and minimizes weight

- cut: an assignment of a graph's vertices into two non empty lists
- cut property: any minimum weight crossing edge must be in the MST
- crossing edge: edge connecting two cuts together

Prims Algorithm:

1. Create set of visited nodes
2. Visite nodes from some starting point by following minimum edge weight

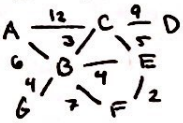


$\overline{AB}$ $\overline{BD}$ $\overline{BC}$ $\overline{CE}$ $\overline{EF}$

* break ties alphabetically
* keep track of traveled edge weight distance, you minimize on those

Kruskal's Algorithm

1. sort edges by weight
2. build MST from list



$\overline{EF}$ $\overline{BC}$ $\overline{BE}$ $\overline{B6}$ $\overline{AB}$ $\overline{CD}$

Primitives: Byte, short, Char, Int, Float, Double, Long, Boolean

Equals (=) operator copies bits of b to a (a=b)
Primitives are copied by normal way, Ref. types have pointers copied

Pass-by-Value

Array: int[] arr = new int[size]

class methods: static
Instance: must create instance of class

Static Type: everything in Java but be explicitly given a type

git status: uncommitted changes
git log: gives commit log

Insertion Sort:

- Process
  1. Iterate over array
  2. For-each item insert it into correct spot

```
for(e: arr)
    for(j=i  j>0 and arr[j] < arr[j-1])
        swap(arr, j, j-1)
```

- Runtime
  - Best: $\Theta(N)$
  - Worst: $\Theta(N^2)$

Notes:
- efficient for small number of elements
- good on almost sorted arrays

Pros:
- easy to implement
- Ordered sequence are closer to $\Theta(N)$

Cons:
- not suitable for large data sets
- polynomail at worst case

Selection Sort:

- Process
  1. Select smallest item
  2. move to start
  3. Repeat
- Runtime
  - Always $\Theta(N^2)$

Notes:
- always $\Theta(N^2)$

Heap Sort:

Process
1. Build heap (heapify)
2. Pop out items into list

- Runtime:
  - $\Theta(N \log N)$
  - Gets outperformed by other NlogN algorithms

Notes:
- Good for space-tight systems
- Sorts in place

pros:
- $\Theta(N \log N)$
- Can be impl. in place

Cons:
- not as fast
- unstable
- small-medium

Quick Sort:

Process:
1. Split collection to be sorted into three collection around pivot (smaller, eq, greater)
2. Recursively call quicksort on collections
3. merge sorted collection by concatenation

- Runtime: $\Theta(N \log N)$ but technically $\Theta(N^2)$

Notes:
- $\Theta(N^2)$ but in real world runs NlogN

pros:
- executes in place
- fast

Cons:
- unstable
- can be polynomail

Merge Sort:

- Process
  1. split collection in half
  2. run merge sort on these collections
  3. merge sorted halves

  * element list size of 0 or 1 is sorted
- Runtime
  - $\Theta(N \log N)$ worst case

Notes:
- space inefficiency

pros:
- good choice when data is fetched off other memory
- runtime

Cons:
- overhead cost of copying data
- space

Counting Sorts:

- Require at least NlogN comparisons to sort N elements
- Radix → number of values a digit can take on e.g bin=2
- Radix sort checks/sorts elements in passes

LSD (Least Significant Digit):
1. sort on least important key
2. sort on next key
3. requires stable sort

MSD: uses the msd instead. Implemented in a similar way

Runtime: W → word size
Performance → $\Theta(WN)$
Space → $\Theta(W + N)$

BucketSort:
- if your word size is k, make k buckets to sort with for $\Theta(1)$ sort

Common Data Structure Runtimes: (Avg: Worst)

	Access	Search	Insertion	Deletion	Space
Array	$\Theta(1)$	N	N	N	N
Stack	N	N	1	1	N
Queue	N	N	1	1	N
LinkList	N	N	1	1	N
HashTable	-	1/N	1:N	1:N	N
BST	logN:N	logN:N	logN:N	logN:N	N
BTree/RB	logN	logN	logN	logN	N

Array Sorting Algorithm:

	Best	Avg.	Worst	Space
Quicksort	NlogN	NlogN	N^2	logN

- Best: pivot divides array in equal halves by coming in the middle → Random Order
- Worst: List arranged ascending/descending order

Mergesort NlogN " " N

- Best Case: least number of comparisons
- Worst case: max num. of comparison usually when every element needs to be compared when merging

Heap Sort: NlogN " " $\Theta(1)$

- Best: you have same NlogN on both best/worst but there is going to be a logN·N and blogN·N where a<b

Insertion Sort: N N^2 N^2 1

- best: Almost sorted list
- worst: Random list

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

Runtime Analysis:

```
i(int N):
    TreeSet t;
    for( N =: i  i>0   i /= 2)
        t.add(i) → log N
```

$\Theta := (\log N \cdot \log(\log N))$

you want to look at this as you insert takes logN time and you have N option, you size is logN so $K \log k$ $k = \log N$