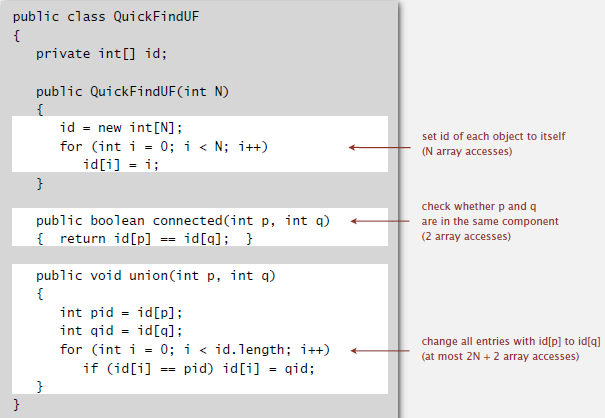
**Union-find**

1. **Quick-find** (eager approach)

Data structure: Integer array id[] of length N. p and q are connected iff they have the same id.

Find: Check if p and q have the same id.

Union: To merge components containing p and q, change all entries whose id equals id[p] to id[q].



Quick-find is too slow: initialize(N) + union(N) + find(1)

* Union is too expensive: N array access -> takes N^2 (quadratic) array accesses to process a sequence of N union commands on N objects.

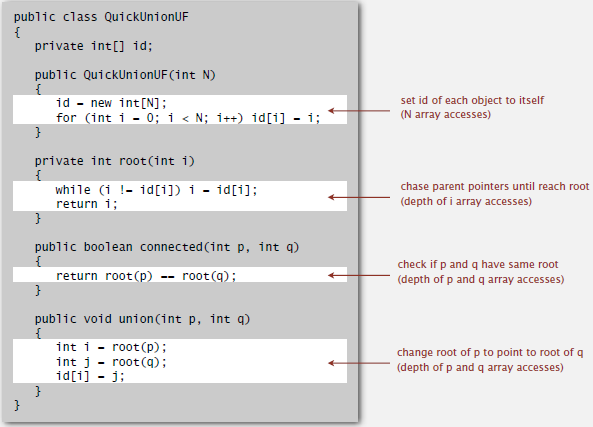
1. **Quick-union** (lazy approach)

Data structure: Integer array id[] of length N. id[i] is parent of i.

Find: Check if p and q have the same root.

Union: To merge components containing p and q, set the id of p's root to the id of q's root.

* only one value changes



Quick-union is also too slow: initialize(N) + union(N, including cost of finding roots) + find(N, worst case)

* Trees can get tall: Find too expensive (could be N array accesses).

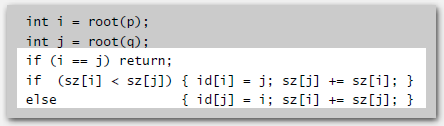
Improvement:

1. Weighted quick-union: Keep track of size of each tree (number of objects). Balance by **linking root of smaller tree to root of larger tree**.

Data structure: Same as quick-union, but maintain extra array sz[i] to count number of objects in the tree rooted at i.

Find: identical to quick union. -> takes time proportional to depth of p and q.

Union: Link root of smaller tree to root of larger tree. Update the sz[] array. -> takes constant time, given roots.



Depth of any node x is at most lg N. (base-2 log)

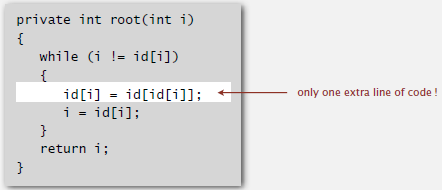
-> Depth Increases by 1 when tree T1 containing x is merged into another tree T2.

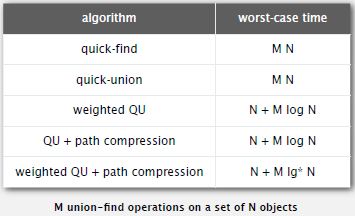
The size of the tree containing x at least doubles since |T2|≥|T1|.

Size of tree containing x can double at most lg N times.

1. Path compression: **After computing the root of p, set the id of each examined node to point to that root**. -> keep tree flat

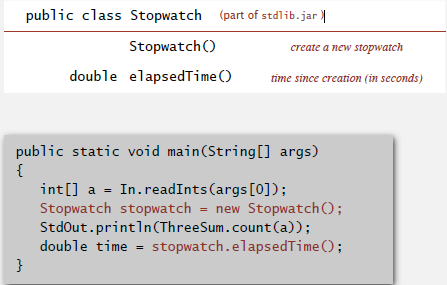
Implementation: Make every other node in path point to its grandparent (thereby halving path length).



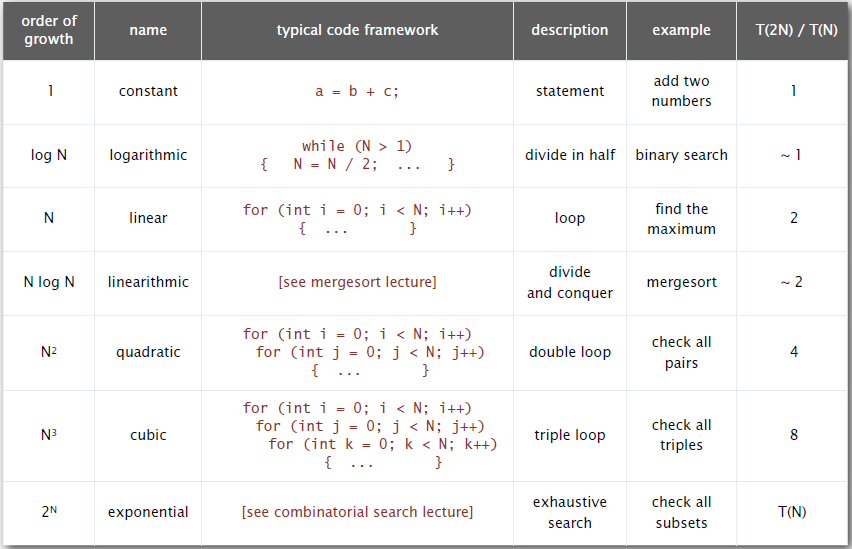
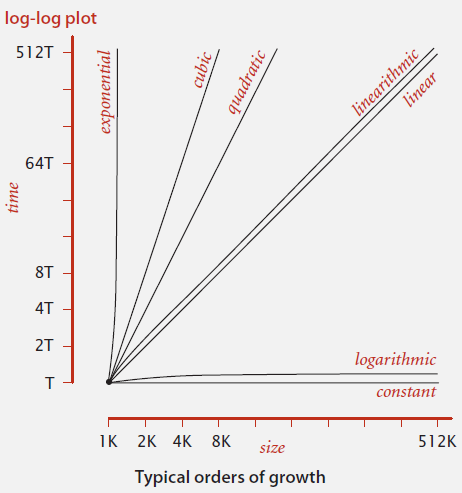
 (lg\*N: times of lg operations on N to reach 1, lg\*(2^65536)=5)

**Analysis of Algorithm**

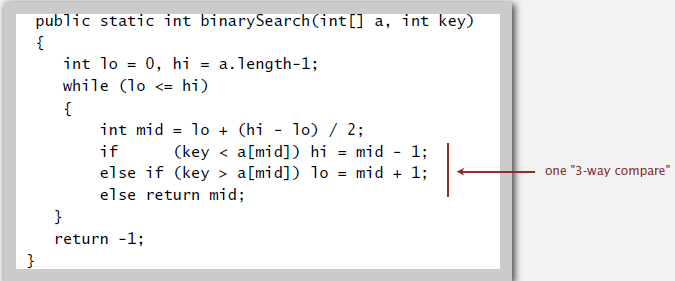
1. Measuring the running time:



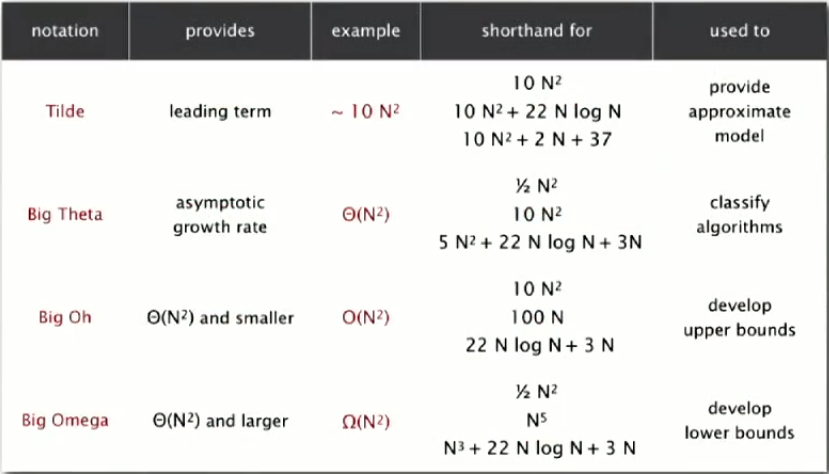
1. Order of growth



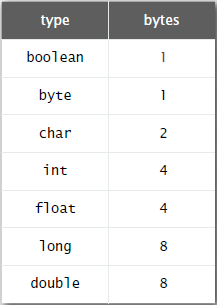
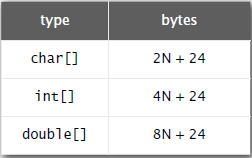
e.g. Binary search (logN)



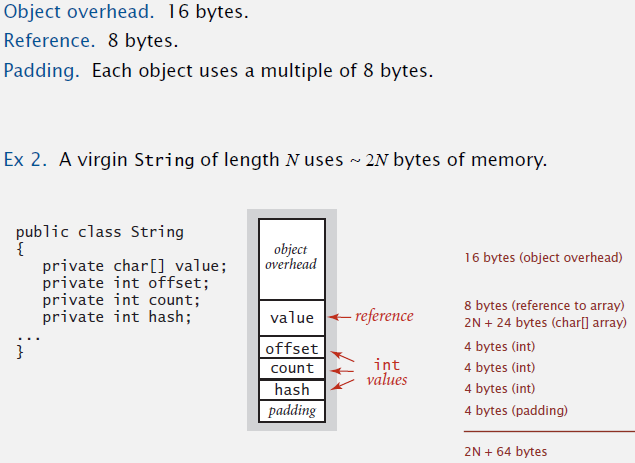
1. Notations



1. Memory

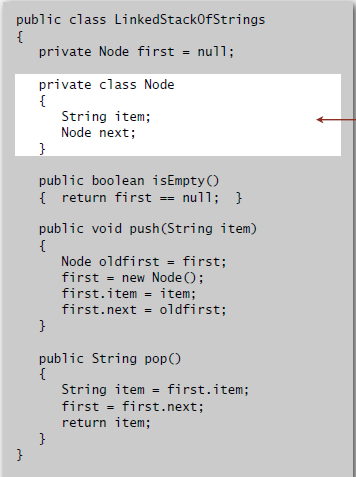
Object in Java:



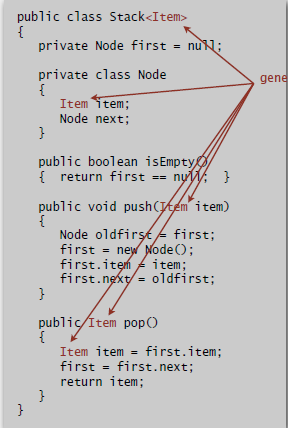
**Stacks and queues**

1. Stack

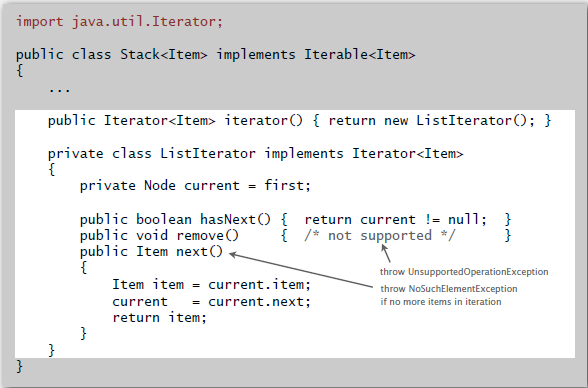
Linked-list implementation:



**Generic** stack: stack of different types



Stack iterator:

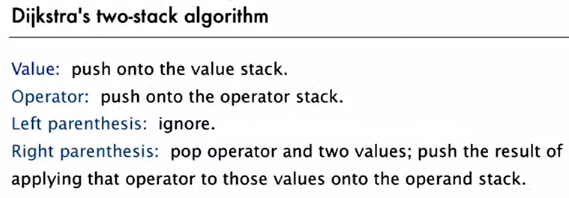


Stack applications:

1. Function call: push local environment and return address.

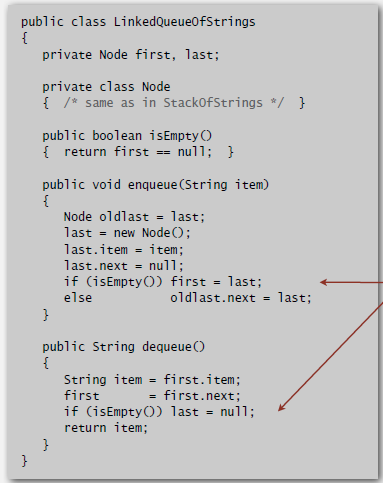
Return: pop return address and local environment.

1. Arithmetic expression -> interpreter



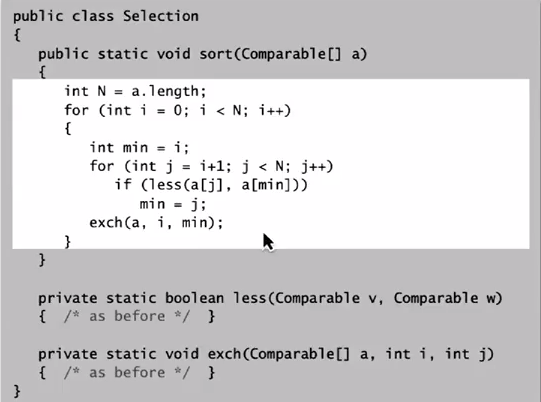
1. Queue

Linked-list implementation:

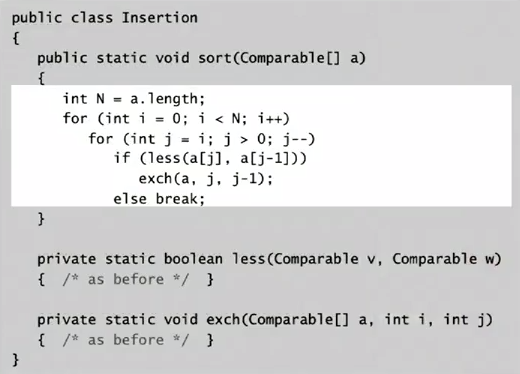


**Sorts**

1. Selection sort: **N2**/2 compares and N exchanges -> even if input is sorted

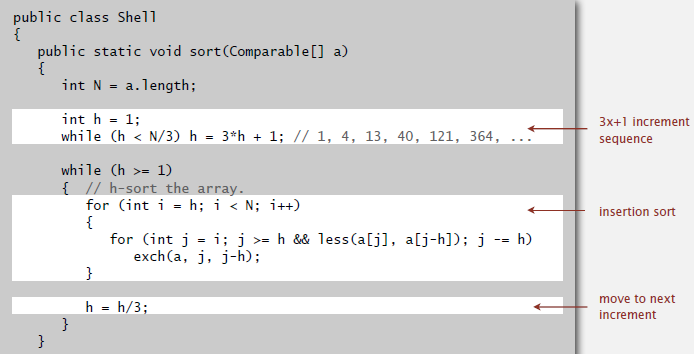


1. Insertion sort: **N2**/4 compares and N2/4 exchanges on average



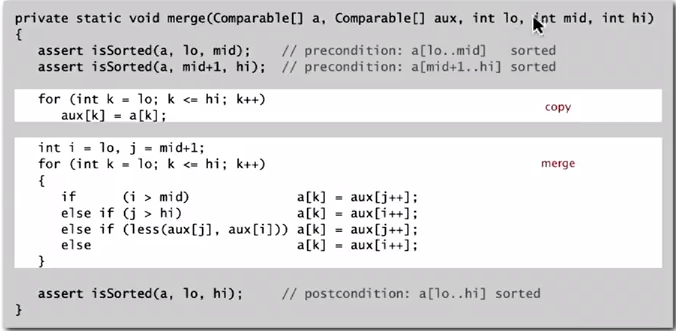
1. Shell sort: **NlogN** compares

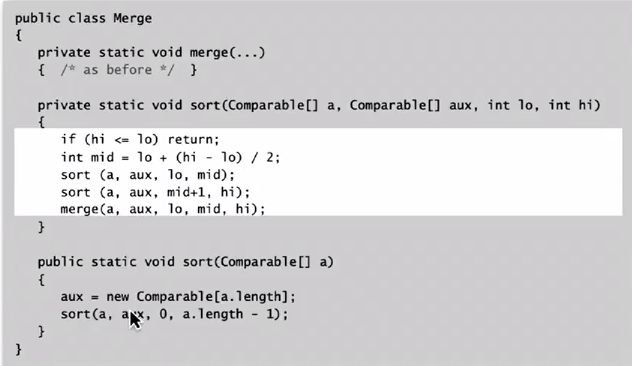
Insertion sort with stride length h



1. Merge sort (Stable, guarantee **nlogn** performance)

* At most Nlgn compares (Between ½NlgN and NlgN)





Assertions: statements to test assumptions about your problem

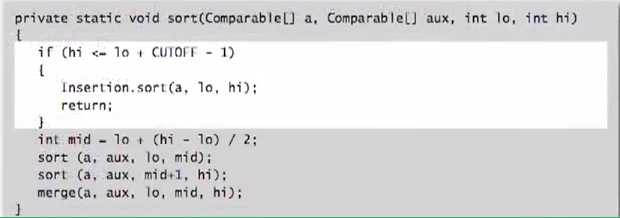
java –ea MyProgram //enable assertions

java –da MyProgram //disable assertions

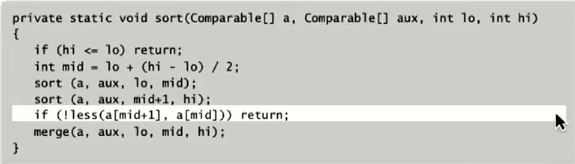
Pratical improvements:

1. use insertion for small subarrays

-> cutoff to insertion sort for -7 items

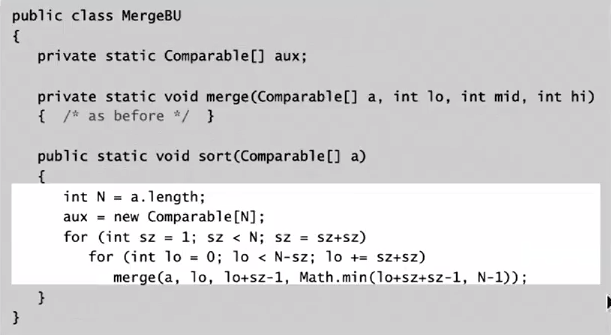


1. Stop when the biggest item in first-half <= smallest item in the second half



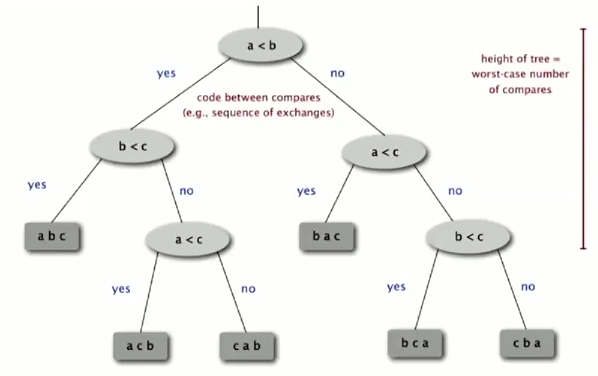
1. Bottom-up mergesort: no recursion needed

Merge from subarrays of size 1 to 2,4,8…



**Any compare-based sorting algorithm must use at least lg(N!)~NlgN in the worst case**

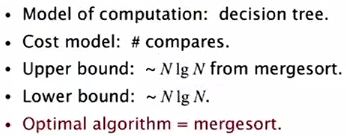
Decision tree: height -> worst-case no. of compares



N distinct values - > N! different orderings

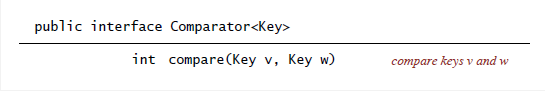
Binary tree of height h: 2h leaves -> h ≥ lg(N!) ~ NlgN

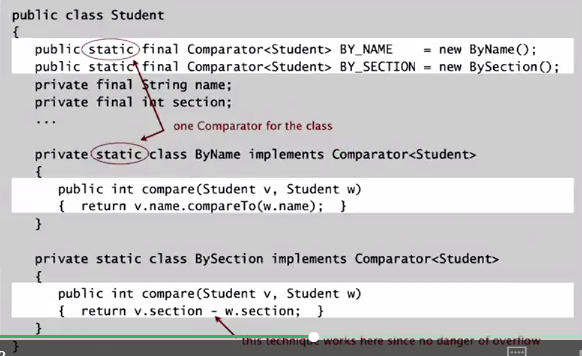
Complexity of sorting:



* Merge sort is optimal with respect to no. of compares, but not with respect to space usage

**Comparator** interface:

****



Stable sort: preserve the relative order of items with equal keys

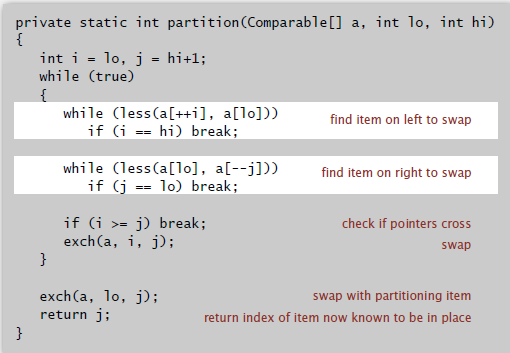
* Insertion sort and merge sort are stable

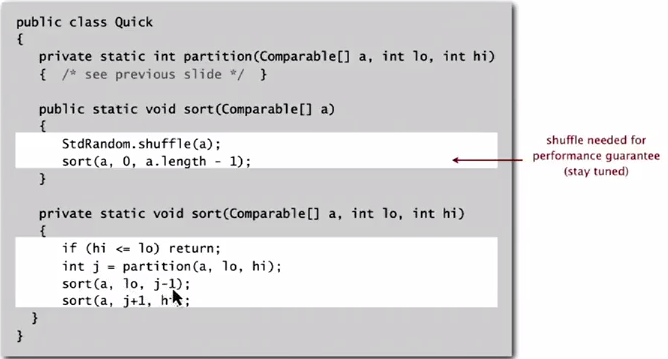
Equal items never past each other

* Selection sort and shell sort are not stale

Long distance exchange might move an item past an equal item

1. Quick sort





Best case: ~NlgN

Worst case(sorted/reverse sorted or has duplicates): ~N2/2

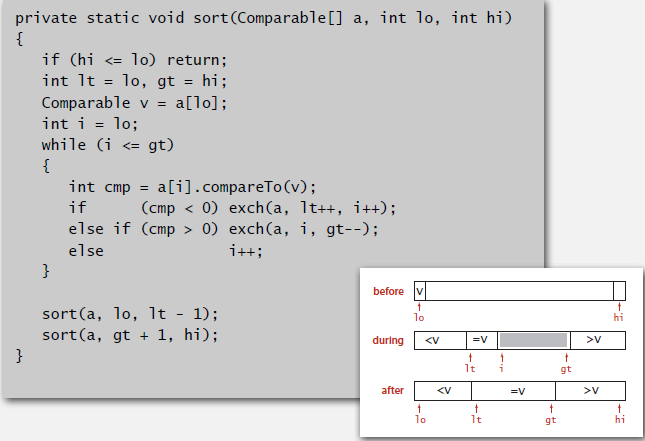
Average case: ~1.39NlgN -> 39% more compares than mergesort, but faster than mergesort because of less data movement.

Not stable.

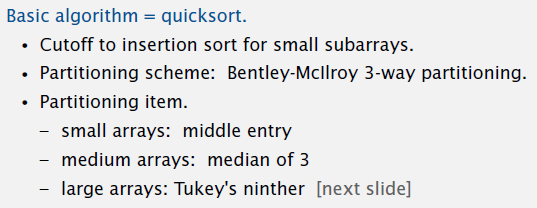
Practical improvement:

1. Cutoff to insertion for 10 items
2. Choose the median of 3 sample items as pivot
3. 3-way partitioning

Large number of duplicate keys: quick sort goes ~N2/2



System sort:



Java system sorts:

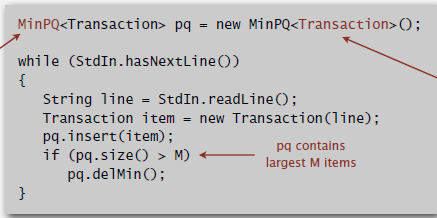
import java.util.Arrays;

Array.sort(a); -> use tuned quicksort for primitives types; tuned mergesort for objects

1. Heapsort

**Priority queue**: remove the largest/smallest item

e.g Find the largest M items in a stream of N items



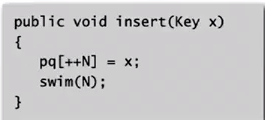
* pq keeps track of the largest M items

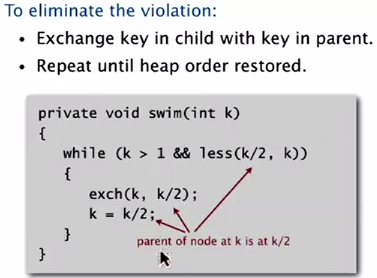
**Binary heap**

1. Parent’s key no small than children’s key
2. Parent of node k is k/2

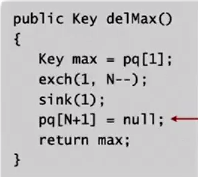
Children of node k are 2k and 2k+1

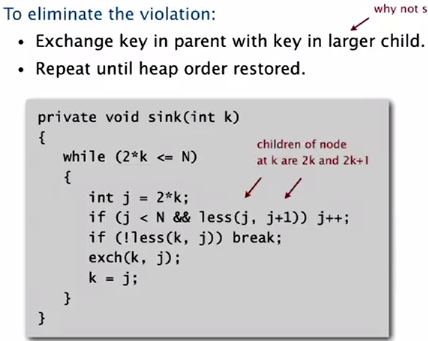
1. Insertion: add note at end, then swim it up (≤1+**lgN** compares)





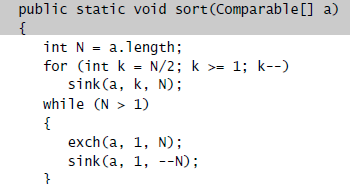
1. Remove (delete the maximum): Exchange root with node at end, then sink it down (≤2**lgN** compares)





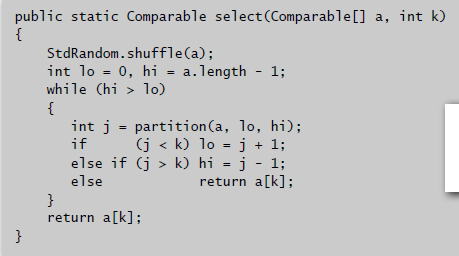
**Heapsort** (in-place with **NlgN** guarantee, not stable):

1. Create max-heap using bottom-up method(≤2**N** compares and exchanges)
2. Repeatedly remove the largest item(≤2**NlgN** compares and exchanges)



1. Selection: Given an array of N items, find a kth smallest item.

Quick-select:

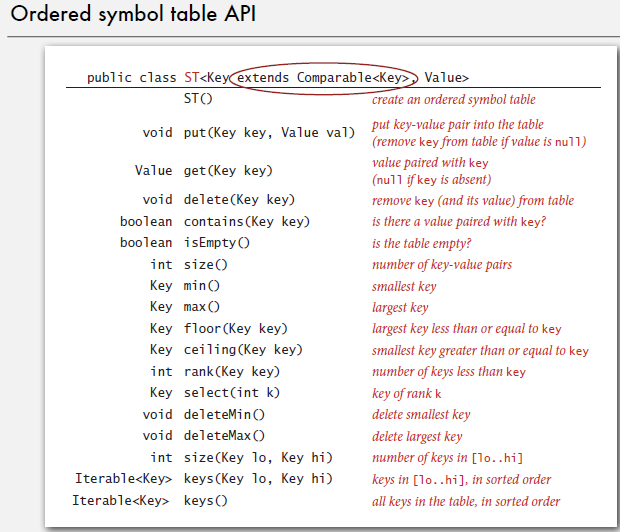


Average: ~N

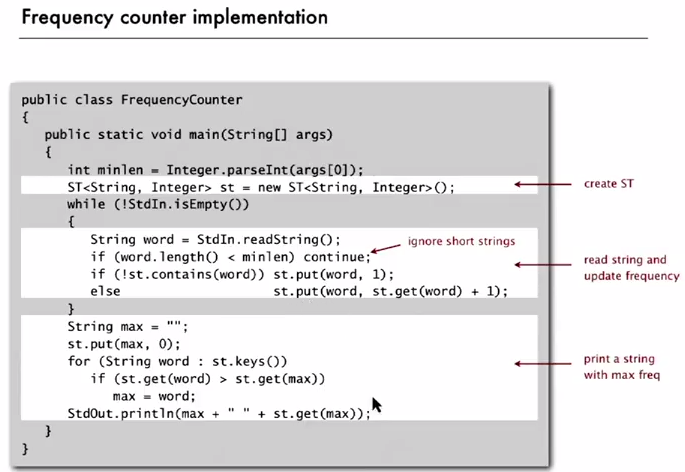
Worst case: ~N2/2

**Symbol tables**

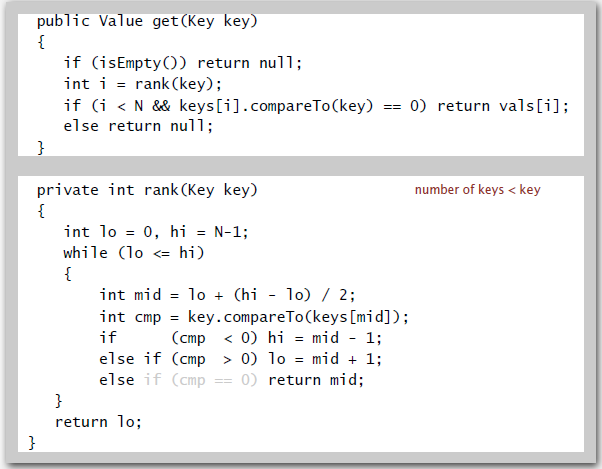
1. Ordered ST API



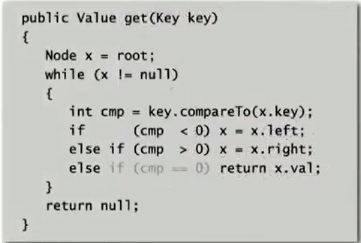
Frequency counter:



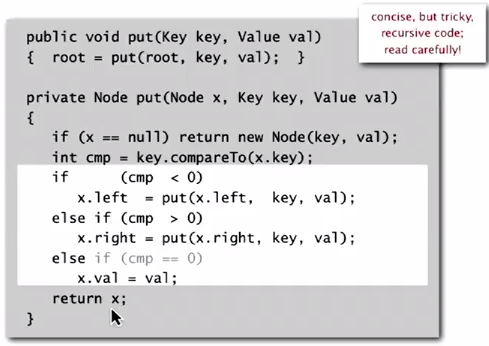
1. Binary search for ST: maintain a ordered array for key-value pairs



1. **Binary Search Tree**: each node’s key is larger than all keys in its left subtree; smaller than all keys in its right subtree
2. Get key (No. of compares: 1+depth of tree):

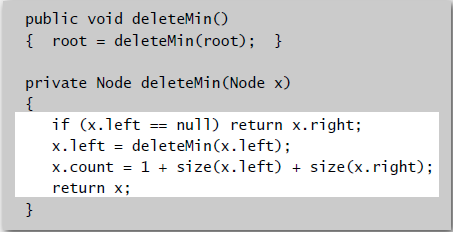


1. Add value (No. of compares: 1+depth of tree):

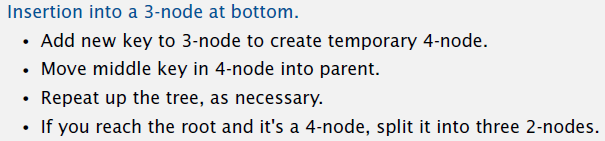


If N distinct keys are inserted into BST in random order, the expected no. of compares for search/insert is ~2lnN.

1. Delete minimum



1. Balanced Search Tree
2. 2-3 Search Tree: Perfect balance -> Every path from root to null link has same length.

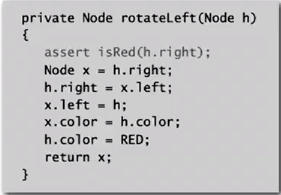


1. Left-leaning Red-black BST:

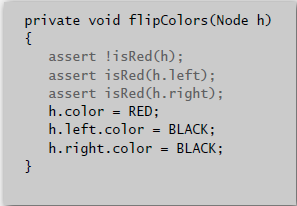
* No node has two red links connected to it (never two red links in a row)
* Every path from root to null link has the same number of black links
* Red links lean left

-> Height of tree <= 2lgN

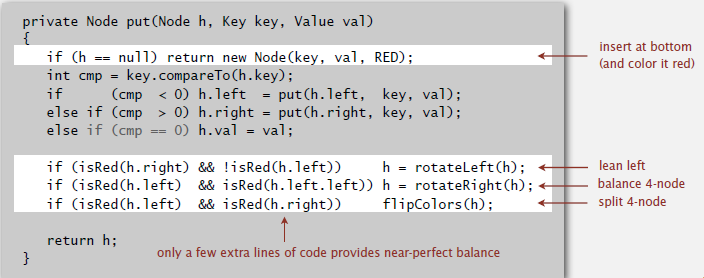
Left rotation:

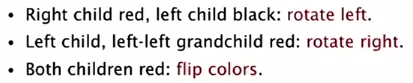


Flip colors: when both children are red, flip the color to parent

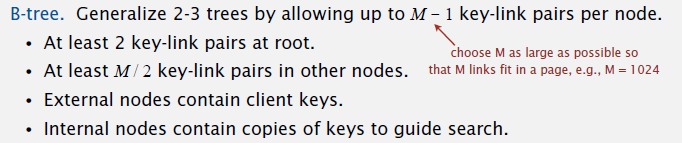


Insertion:



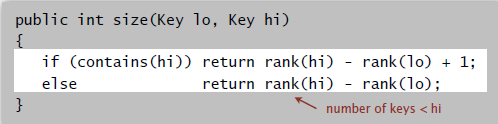


1. B Tree, B+ Tree, B\* Tree, B# Tree

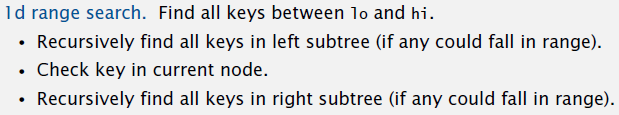


A search or an insertion in a B-tree of order M with N keys requires at most logM/2N probes.

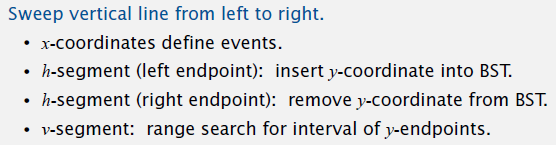
1. Geometric application of BSTS
2. 1d range count: How many keys between lo and hi ?



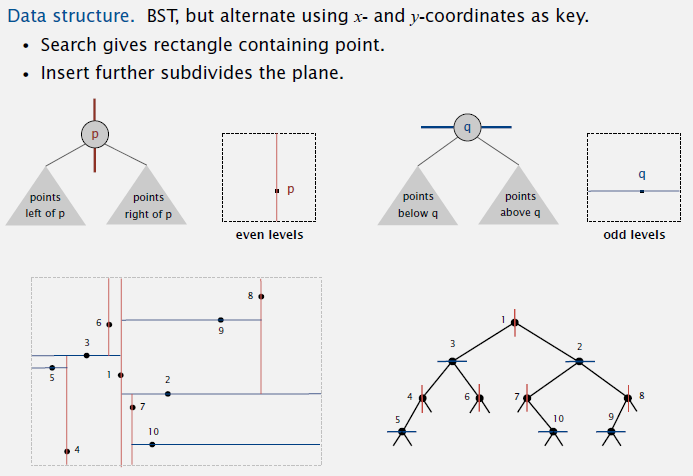
1d range search: Find all keys between lo and hi.



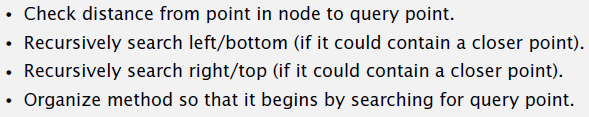
1. Orthogonal line segment intersection: sweep-line algorithm



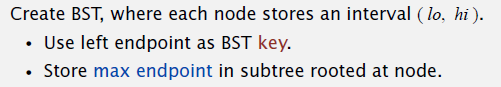
1. 2d tree construction: Recursively partition plane into two halfplanes.



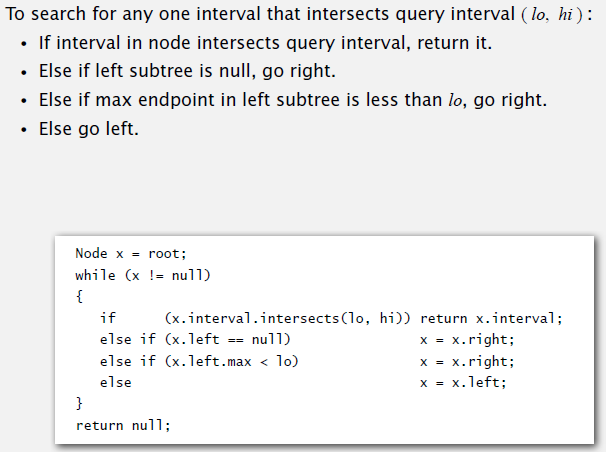
Nearest neighbor search in a 2d tree:



1. Interval Search Trees



Search:



Orthogonal rectangle intersection: sweep-line algorithm

