3.14.2020

* Dynamic Connectivity

Find connections----computer network, photo pixels

Assumption: connected to itself; transitive; symmetric

Connected components

* Quick Find

(eager algo; using index array)

The interpretation is the two objects, p and q are connected if and only if, their entries in the array are the same.

An array of size N; elements should be the first number in the connected components

Union too expensive!

* Quick Union

Integer array id[ ] of size N

Then id[i] is parent of I (What in elements is not entry but parent)

Find: check p and q have same root, if not;

Union: change p’s root to a child of q’s root

Find root:

While ( i i=id[i]) # find i until id of i is itself i=id[i] # if not equal just move up one level;

return i

But tree can get too tall

* Improvement

**Weighting**

(make sure link small tree to larger tree by tracking number of objects in the tree)

1. Running time:

Find: proportional to depth of p and q

Union: constant time given roots

N+M log N

1. Depth of any node x is at most log2N

**Path Compression**

(After getting the root of p, set id of p directly to the root)

N objects, M union operations

1. Running time: maximum c(N+M log\*N) with weighting
2. Log\*N: number of times you need to take the log of N to get 1 (most cases should be less than 5)

No linear algorithm for quick union!

* Percolation

Settings:

1. N-by-N grid of sites
2. Each site is open with prob p and blocked with prob 1-p
3. System percolates if top and bottom are connected by open sites

There is a threshold p\*,

If p>p\*, almost certainly percolates

If p<p\*, almost certainly does not percolates

Use simulation to find p\*:

1. All blocked
2. Declare random sites open until percolates
3. Vacancy percentage estimates p\*

3.15.2020

* Binary Search

Find index of the key in a sorted array:

1. Left pointer (LP)
2. Right pointer (RP)
3. If LP=RP but not the key: key is not in the array
4. At most 1+log N compares

T(N) <= T(N/2)+1 <= T(N/4)+1+1<=…..<= T(N/N)+1+1+….+1=1+logN

* Three sum

Solution 1: N2 for insertion sort, log N for binary search

1. Sort the array
2. For each pair of a[i] and a[j], binary search for –(a[i]+a[j])
3. Only count if a[i] < a[j] < a[k]

3.18.2020

* Stacks

Linked list: (each node contains an item and a link; the link should be the pointer to the next node)

Array: you need to declare the size of array

Overflow: use resizing array for array implementation

Underflow: throw exception if pop from an empty stack

Loitering: holding a reference to an object when it is no longer needed

Resizing Array

1. Grow array push()

If array is full, create a new array of twice the size, and copy items (inserting first N items takes time proportional to N, not N2)

N+(2+4+8+…+N) ~ 3N

1. Shrink array pop()

Have size of array when array is one-quarter full

Linked list: guaranteed constant operations for push and pop

Resizing array: totally will be less operation comparing to linked list

For linked list, you need linear time to access the last node; but constant time for array.

* Generics and Iterators

Eg: Stack<Apple> s = new Stack<Apple>();

Try to avoid type casting!

Iterator:

Has methods hasNext() and next()

For data structure that is iterable

Eg: for (String s : stack) # for each string in the stack

StdOut.println(s)

* Stack applications

How a complier implements a function:

Function call: push local env and return address

Return: pop return address and local env

Dijkstra’s Two-stack algorithm

3.20.2020

* Callbacks : reference to executable code
* Selection sort

in iteration I, find index min of smallest remaining entry

then swap a[i] and and a[min]

(N-1)+(N-2)+...+1+0 ~ N^2/2 compares

and N exchanges

And use quadratic time even if input is sorted!

* Insertion sort (N^2)

in iteration I, swap a[i] with each larger entry to existing array number

N^2/4 compares and N^2/4 exchanges on average

about twice faster than selection sort.

Best case & Worst case

1. already sorted : N-1 compares and exchages

2. reverse order : N^2/2 compares and N^2/2 exchanges

Def. Inversion: a pair of keys that are out of order (逆序数）

Def. An array is partially sorted if the number of inversions is <= cN

Then : For partially sorted arrays, insertion sort runs in linear time

( Number of exchanges equals the number of inversions; number of compares = exchages + (N-1))

* Shellsort

(h-sorted subsequences)

确定一个间隔h，先以此间隔上的数进行排序；然后减小这个h，再次以新的h来排序，直到最后h变为1,成为最后的普通的insertion sort

usually we start h= 3n+1; n is an integer

But the worst case number of compares with 3n+1 increment is O()

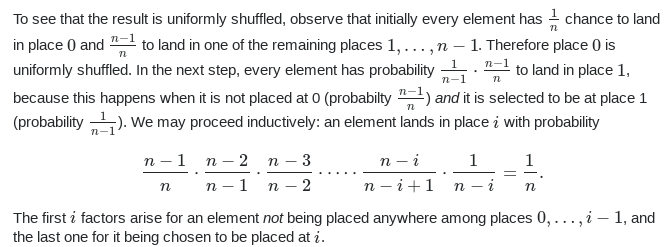
* Shuffling

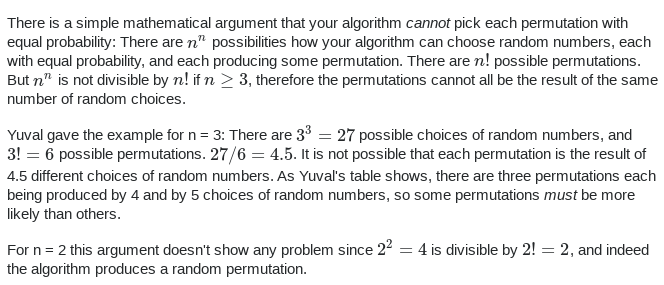
in iteration a, pick integer r between 0 and a uniformly at random

then swap list[a] and list[r]

----only linear time

PS: can not pick integer between 0 and N-1 (won’t be uniformly random shuffling)





* Convex Hull

The smallest perimeter fence enclosing all the N points

1. Graham scan demo

choose point p which has smallest y-coordinate

sort points by polar angle with p

consider points in order; discard unless it create a couterclockwise(ccw) turn

How to check ccw?

Slope (infinity? Positive? Negative?)

or k=(bx-ax)(cy-ay)-(by-ay)(cx-ax)

if k=0, a,b,c colinear

if k>0, ccw

if k<0, clock wise

3.23.2020

* Mergesort(NlogN)

1. Two parts, two pointers;

2. when equal, take the first;

java assertion:

java -ea MyProgram //enable assertion

java -da MyProgram//disable assertion

Memory:

use extra space proportional to N

A sorting algorithm is in-place if it uses<= clogN extra memory

Improvement for mergesort:

1. transfer to insertion sort if length of subarray is less than 7

2. If the biggest item in the first half is smaller than the smallest item in the second half; merge them directly

* Bottom-Up Merge sort

No need for recursion

get sorted with sub-arrays with size 2,4,8,….

using extra space!

* Model computation complexity

Merge sort upper bound : N log N

lower bound: N log N

Merge sort is an optimal algorithm wrt compares time, but not space

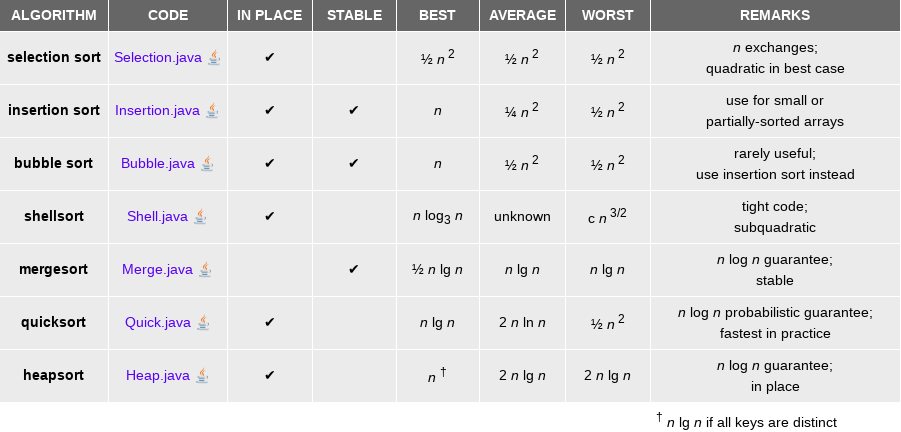
For an array of length N:

Binary tree of height has at most leaves;

the array has N! different orderings for N items;

Therefore need at least N! leaves

* Cheat sheet



* Comparator interface
* Stability:

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Some sorting algorithms are stable by nature like Insertion sort, merge sort

3.24.2020

* Quicksort

1. shuffle the array

2. partition so that, for some j

-entry a[j] is in place

- no larger entry to the left of j

- no smaller entry to the right of j

3. Sort each piece recursively

Do not require extra space

* Average case analysis of quicksort

average number of compares of N distinct keys, which satisfies :

where each 1/N is the probability of such partitioning

Multiply two sides by N and subtract same equation for N-1

approximate this sum by an integral:

1. worst case will involve quadratic compares

2. 39% more compares than mergesort, but faster than mergesort in practice because of less data movement

3. in place method

4. quick sort not stable

* Quick sort improvement

1. insertion sort for small arrays

2. sample the item and take median of sample as the first partitioning number

* Selection: find kth largest (top k)

Quick select: Do partition, until number of elements on the left equals to k

quick select takes linear time on average

* For duplicate keys

Mergesort with duplicate keys: Always between 0.5 NlogN and N logN

Quicksort with duplicate keys: goes quadratic unless partioning stops on equal keys!

3-way paritioning:

middle area contains all elements equal to pivot.

3.27.2020

* Priority Queues

Stack: remove the item most recently added

Queue: remove the item least recently added

Priority queue: remove the largest (or smallest) item

* Application: always keep top M largest records using priority queue
* Binary Heap:

1. complete tree (height of tree with N nodes should be log2N

2. Binary heap: array representation of a heap-ordered complete binary tree

(Keys in nodes; parent’s key no smaller than children’s keys)

Array representation:

* + - starts at 1 (not zero!)
    - Take nodes in level order (left to right)
* Property of binary heap
  + 1. largest key is a[1]
  + 2. Parent of node at k is at k/2 (java会向下取整，保证第一个node是1不是0)
  + 3. Children of node at k are at 2k and 2k+1

Scenario A: child’s key becomes larger than its parents’ keys

To eliminate the violation: Swim(int k)

* + 1. Exchange child and parent
  + 2. Repeat until heap order resorted (while loop)

To insert :

* + 1. Add node at end and them swim it up
  + 2. at most 1+log2N  compares (which will go to the root)

Scenario B: parent’s key becomes smaller than one (or both) of its children

To eliminate the violation: Sink(int k)

* + 1. Exchange **larger** child and parent
  + 2. Repeat until heap order resorted (while loop)

To delete the maximum:

* + 1. Exchange root with node at end, then sink the new root down
  + 2. at most 2log2N compares (which will go to the root)

Improvements:

1. d-array heap

2. Fibonacci heap

* Underflow and overflow

underflow: delete from empty PQ

overflow: add no-arg constructor and use resizing array

* Heapsort(NlogN in-place sort, but not stable)

1. create max-heap with all N keys

2. repeatedly remove the maximum key

Build heap using bottom-up method:

for (int k=N/2; k>=1;k--) (倒数第二层开始sink)

sink(a,k,N)

Heapsort guaranteed NlogN for worst cases, but no stable and longer inner loop

Mergesort: guaranteed NlogN for worst cases but need linear extra space

Quicksort: quadratic time in worst cases

3.28.20200

* 2-3 Trees

Allow one or two keys per node

one key, two children; two keys, three children

Worst cases: lg N

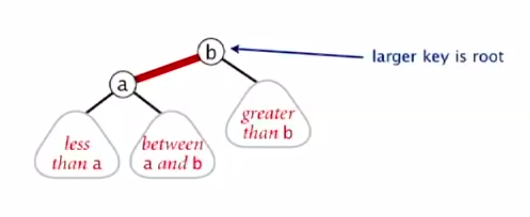
best cases: log3N

Guaranteed logarithmic operation for search, insert, delete

But: complicated to analyze and handle splitting

* Red-black tree BST

Idea: the larger key in the 2-3 tree node becomes the root:



Property:

1. No node has two red links

2. Every path from root to null link has the same number of black links

3. red links lean left

Search is same as elementary BST

For insertion:

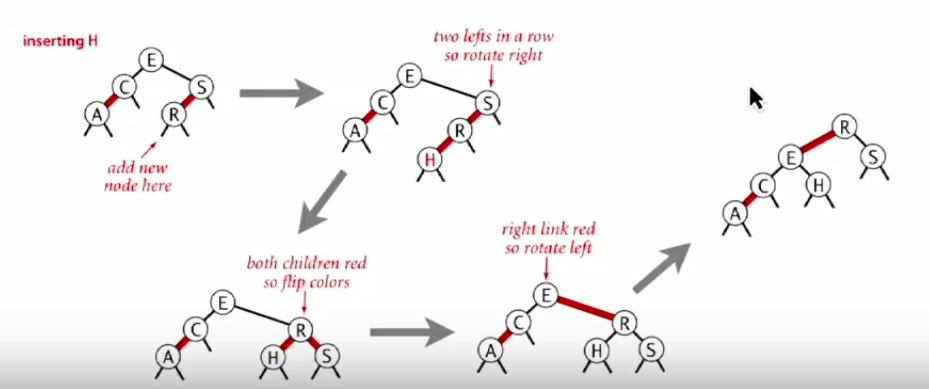
1. Insert like standard BST, color the new link as red

2. If new red link is a right link, rotate left

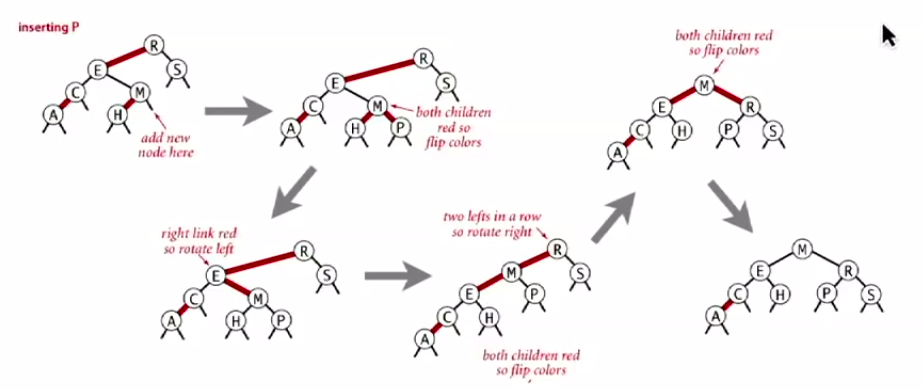
3. If new red link is a right link and parent becomes two red linked; flipped both red link to black

4. If new red link is a left link but cause two red left link in a line; rotate right and then flip as 3.

eg1:



eg2:

Height of tree is less than 2 lgN in the worst case

4.3.2020

Geometric apps for BST

* 1d range search

(how many keys between low and high?)

* Line intersection search
* KD tree(multi-dimensional partition tree)

2-d orthogonal range search:

partition plane according to a binary tree:

1. for vertical point, left is left, right is right

2. for horizontal point,left is below, right is above

Hash Tables

* Hash function: method for computing array index from key
* Collision: two keys that hash to the same array index
* Uniform hashing assumption:

Each key is equally likely to hash to an integer between 0 and M-1:

1. expect two keys in the same hashing code after ~tosses

2. expect every hash code has at least 1 key after ~MlnM tosses

3. after M tosses, expect most loaded hash code has

* Separate Chaining

(Build a linked list at each index to resolve collision)

Number of probes for search/insert is proportional to N/M

1. Too much wasted space if M is too large

2. List too long if M is too small

* Linear Probing(Open Addressing)

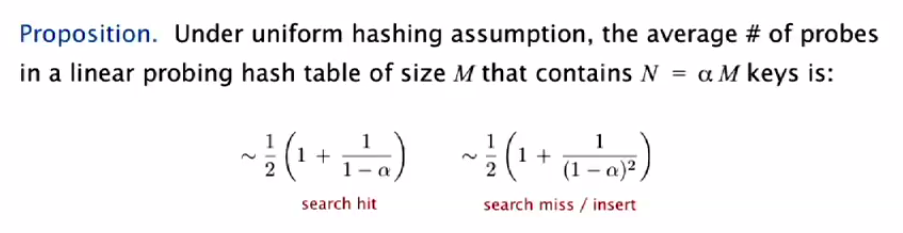
when a new key collides, find next empty slot

1. if we want to search a key not in the table,hash it and find it does not match; keep on searching until we find that there is an empty slot

2. Knuth’s parking Problem:

with M/2 cars, mean displacement is ~3/2

with M cars, mean displacement is ~



Typical choice:

2020.4.5

* Undirected Graph
  + Vertex representation
  + Edge representation:

1. 0-1 matrix (too huge if you have many vertexes)

2. adjacency-list (vertex as index, indicate an array list of its adjacent vertexes)

* + Real world problem: graph tends to be sparse
* Depth First Search (No same path twice visited)

1. Find all verices connected to a given source vertexes

2. Find a path between two vertices

Algo:

Mark each visited vertex

Return when no unvisited options

DFS marks all verices connected to s in time proportional to the sum of their degrees

After DFS, can find vertices connected to s in constant time and can find a path to s in time proportional to its length

* Breadth First Search

Repeat until queue is empty:

1. Remove vertex v form queue

2. Add to queue all unmarked vertices adjacent to v and mark them

BFS computes shortest path(fewest number of edges) form s to all other verices in a graph in time proportional to E+V