## Introduction

API = Application Programming Interface

### Dictionary Problem

The API:

Data: A set of key/value pairs, with at most one pair per key (i.e., no two keys are equal)

Operations:

Query: Find the value associated with a given key or report that the key is not there.

Update: Store a value with a given key.

Approaches:

Hashing schemes (faster to lookup)

Balanced binary search tree (avoid collisions, faster to know the neighbors)

### Network Routing along Paths with Maximum Bandwidth

Compute a maximum spanning tree ⇒ Kruskal’s Algorithm

API

Data: Partition of a set of objects into disjoint sets

Update: Merge two sets by forming their union.

Query: Do two objects belong to the same set?

Analysis: n nodes and m edges

sort m items

perform m queries

n-1 merge operations

### Verify Correctness of Geographic Information System data

Given a set of n line segments, check that no two intersect ⇒ Line sweep algorithm

vertical line sweeping from left to right.

cross = red, not cross = black

each line has two operations (turn red and turn black)

order the red lines from top to bottom

If red lines never cross ⇐ ordering of the red lines doesn't change

if red lines cross ⇐ intersected lines are neighbors + ordering of the intersected lines will sweep

keep track of neighbor red lines and if their ordering changes

API

Data: Sorted set of objects, ordered by pairwise comparison

Updates:Insert/Remove an object

Queries: Find next or previous object in sorted order

Data Structure: balanced binary search tree

O(log n) per update/query

crossing detection in O(n log n) time

### Computer Gaming

shortest path in a graph ⇒ Dijkstra’s algorithm (from starting point, update the shortest path to all vertices)

API

Data: set S prioritized by D[v]

Operations: find and remove item with D[v]; decrease D[v]

Analysis for n vertices and m edges

(1) find and remove minimum <= n

(2) decrease priority <= m

with binary heap ⇒ O(log n) operations ⇒ O(m log n) for Dijkstra

m > n ⇒ we can accept slower (1) and faster (2)

Steps

associate extra information with input object

Kruskal: vertices → set

Segment Crossing Detection: lines → nodes of binary search tree

Dijkstra: vertices → priority queue

Decorator pattern = pattern that allows behavior to be added to an object without affecting the behavior of other objects from same class

## Amortized Analysis

Analysis of Data Structure

Worst-case analysis

for real-time response

overly restrictive ⇐ the overall performance is more important than one case

Average-case analysis

expected value or high-probability bounds for

based on assumption about input OR random choices

usually difficult and inaccurate ⇐ not knowing typical case

Amortized analysis = worst-case time for a sequence of operations

### Toy Example - binary counter for n-bit binary counter

Data: array B of n bits

Operations:

create a new n-bit counter

increment counter

i = 0

while B[i] == 1:

B[i] = 0

i = i + 1

B[i] = 1

Worst case analysis

create = O(n)

increment = O(n) (0111 → 1000)

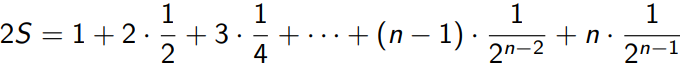
Average case analysis (sometimes not meaning anything)

assume 2^n possible bit patterns are equally likely

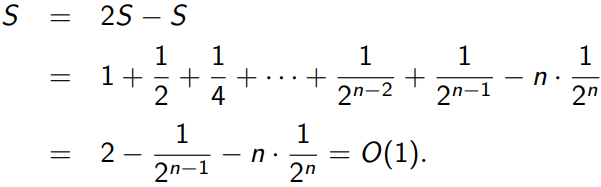
expected time for increment

the probability that you do exactly n operations is 1/2^n





subtracting



Amortized analysis

Tentative definition: for a sequence x of operations,

time per operation of n = O(n) but not really useful

instead, analyze the following

create(n)

repeat k times

increment

⇒ B[0] changes k times

B[1] changes <= k/2 times

B[2] changes <= k/4 times…

⇒ total number of changes <= 2k

Conclusion: running time for algorithm = O(n+k) and create = O(n), increment = O(1)

### Amortized analysis - potential function method

potential function = states of data struction non-negative numbers

potential function = distance between current state and some ideal state

= 0 in initial state, and all other states != 0

amortized time of an operation as

actual(wall clock) time + C()

C = constant of proportionality

we prove upper bounds on the worst-case amortized time required by an operation

max{amortized time(operation, X): X a state of the data structure}

Theorem: for any sequence of operations starting with initial state

total actual time <= total amortized time

Proof: k =length of the sequence

total amortized time =

= total actual time + where

>= total actual time

Ex, for our toy example

ideal state: all bits zero

potential function: number of 1-bits(counting all bits, not just the rightmost bits)

assume that flipping a single bit = C units of time

Create

actual time = O(n)

amortized time = O(n)

Increment

L = number of 1 bits on the right before the first 0

actual time = C\*(L+1)

(decrease by L-1 because we flipped L-1 bits so we r)

amortized time <= C\*(L+1) + C\*(1-L) <= C\*2 = O(1)

### Dynamic Arrays

Operations

Create(n) make a new array of length n

Length returns current len

Set(x, i) store x in location i

Get(i) return item at location i

Increment increase current len by 1

Decrement decrease current len by 1

Using array to implement binary counter

i = 0

while i < B.length and B.get(i) == 1

B.set(0,i)

i = i + 1

if i == B.length: //all bits are 1s

B.increment

B.set(1,i)

Implementation of ArrayList

Data: L = current length

array B of some length |B| >= L //avoid resizing frequently

maintain |B|/4 <= L <= |B|. ideally L = |B|/2

Operations

def create(n):

allocate B of size 2\*n

for i = 0 to 2\*n-1: B[i] = null

L = n

…

def increment():

L = L + 1

if L > sizeof(B):

B\_NEW = new array of size 2\*L

copy B into B\_NEW

set remaining to null

B = B\_NEW

def decrease():

L = L - 1

if 4\*L < sizeof(B):

B\_NEW = new array of size 2\*L

copy B into B\_NEW

set remaining to null

B = B\_NEW

Amortized Analysis

Define

O(1) for set, get, length operations

O(1) amortized time

Create(n)

O(n) actual time

O(n) amortized time

Increment/Decrement when size doesn’t change

O(1) actual time

O(1) for amortized time

Increment (when current size B = L)

Actual time O(L+1) for copying the array and re-assign variable

Amortized time = O(1)

Summary

for all and for initial state

actual time > 0

may be negative

amortized time may be negative (negative ⇒ O(1))

## Hashing Collision

Intro

Dictionary Problem (no two value are mapped to the same key)

Operations:

Search(key) return associated value or throw exception if not found

Set(key, value) adds new pair or update existing pair

Delete(key) removes the pair or throw exception if not found

Hashing

Basics

hash table H with size N > n. = load factor (0 for empty 1 for full)

hash function h : key → indice in hash table

resize H

rebuild data structure when H changes

may choose a new hash function h

Properties/Assumptions

(unrealistic) all equally likely to be accessed

Common approach:

1. Map key to an integer (ex. add up the bytes)

2. Take result modulo some big prime.

3. Take result modulo N.

Operations (problem with collision (two keys are mapped to the same location))

def search(k):

i = h(k)

if H[i] exists return val else exception

def set(k):

store(k,v) in H[h(k)]

### Hash Chaining

Def

each cell of H stores a collection of key-value pairs

Search(k) scan through the collection

…

* extra space for collections and extra time to scan through collection

Ex. chaining

N = 13. h(x) = x mod 13. Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

key 2 5 6 7 8 9

value 41 18 32 59 73 22

44

31

Analysis

time/operation = O(1+length(H[h(k)]))

assumptions

for each key, h(k) is equally distributed over indices

for any two keys, they are independent

**E[time/operation]** = O(1+E[length( H[h(k)] )])

= O(1+E[number of keys colliding with k])

= O(1+)

= O(1+)

**= O(1+)**

* closer to 1 ⇒ more full ⇒ more time to scan and more likely to collide

### Linear Probing

Def

each cell of H stores oe key-value pair

try to store (k,v) in H[h(k)]. If H is full, try in order H(+1), H(+2)...

wrap around mod N

* Single-level store ⇒ space efficient + sequential memory is fast
* Degrades badly when gets closer to 1

Ex. linear probing

N = 13, h(x) = x mod 13

insert 18, 41, 22, 44, 59, 32, 31, 73

key 2 5 6 7 8 9 10 11

value **41** **18** 44 **59** 32 **22** 31 73

Two classical alternative to Linear Probing

Quadratic probing

if H(h) is full, try in order H(h+1), H(h+4), H(h+9)...

Double hashing

add secondary hash function

if H(k) is full, try H[h(k)+1\*h2(k)], H[h(k)+2\*h2(k)], H[h(k)+3\*h2(k)]...

Operations

def search(k):

i = h(k)

while H[i] is nonempty and contains a key != k #infinite loop if full table

i = (i + 1) mod N

if H[i] is nonempty: return its value

else: exception

def set(k,v):

i = h(k)

while H[i] is nonempty and contains a key != k #infinite loop

i = (i + 1) mod N

store (k,v) in H[i]

lazy delete

find k by v and mark the cell “nonempty but unused”

nonempty ⇒ search should not be stopped

unused ⇒ set can use

* slow down followed search

def eager\_delete(k):

i = h(k)

while H[i] is nonempty and contains a key 6= k

i = (i + 1) mod N

if H[i] is empty: exception

j = (i + 1) mod N

while H[j] is nonempty

if h(H[j].key) is not in the (circular) range [i+1..j]:

*#it was supposed to be at position before i+1*

*# when mod, the range should be [i+1:end] union [0:j]*

*#h(H[j].key) = hash the j’s key*

move H[j] to H[i]

i = j

j = j + 1

clear H[i]

Ex. eager\_delete 18

N = 13, h(x) = x mod 13

insert 18, 41, 22, 44, 59, 32, 31, 73

key 2 5 6 7 8 9 10 11

value **41** **18** 44 **59** 32 **22** 31 73

except the bold values, all are wrapped around when they are inserted

⇒ when deleting 18, 44 32 31 73 should be moved

key 2 5 6 7 8 9 10

value **41** 44 32 **59** 31 **22** 73

Analysis of search of Linear Probing

time for search (k) operation <= **length of largest full contiguous block containing H[h(k)]**. Expected value for any random variable V:

E[search\_time]

<= E[largest…H[h(k)]]

Assume there are L different blocks of length L containing H[h(k)]

Pr[any particular contiguous block of length L] =

c > 1 and c depends on load factor

Pr[at least one of L particular contiguous block of L is full]

E[search\_time]

O(1)

depends on the laid factor

some keys take more time than others

Conclusion

linear probing

successful search =

unsuccessful search =

worst case search time is ∼ log n

hashing chaining search = O(1+) ⇒ more gradual increase

worst case search time is ∼ (log n/ log log n)

### Cuckoo Hashing

Def

Two tables . load factor

Two hash functions

Search(k): look in both places ,

Delete(k): look at both tables and clear if found

Set(k, v)

def set(k,v):

t = 0

p = (k,v)

repeat

p ↔

if p is null: return

if cycle detected:

report failure / throw exception [rebuild table]

t = 1 - t

(k,v) = p

Store the pair in

If pair (k’, v’) exists, evict it and store in

May require another pair (k’’, v’’) from and sore it in …

cycle ⇒ fail ⇒ rehash the entire table with different hash functions

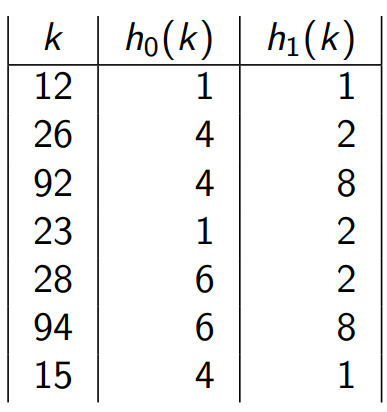
Two approaches of cycle detection

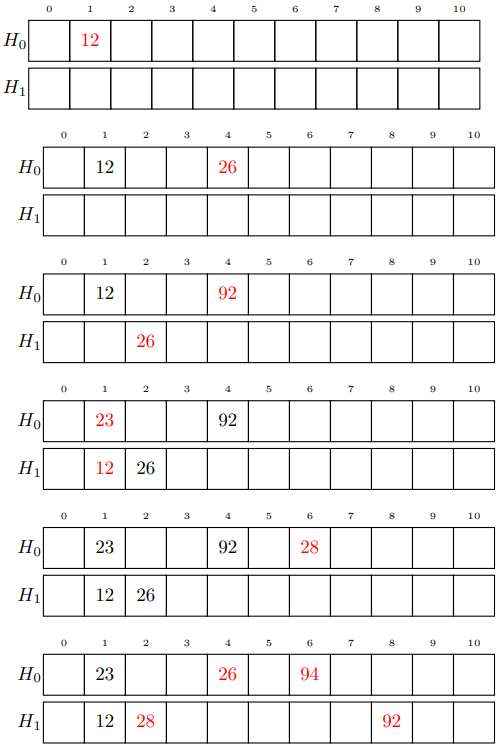
Choose const c as the max value of the loop

c is proportional to the number of tables??

Explicitly check for a cycle

Ex. h0(k) = k mod 11 and h1(k) = (k/11) mod 11





when inserting 94 ⇒ 28 is moved to 2 of H1, 26 is moved to 4 of H0, 92 is moved to 8 of H0

when inserting **15** ⇒ 26 is moved to 2 of H1, 28 is moved to 6 of H0, 94 is moved to 8 of H1, 92 is moved to 4 of H0 where the **15** is placed

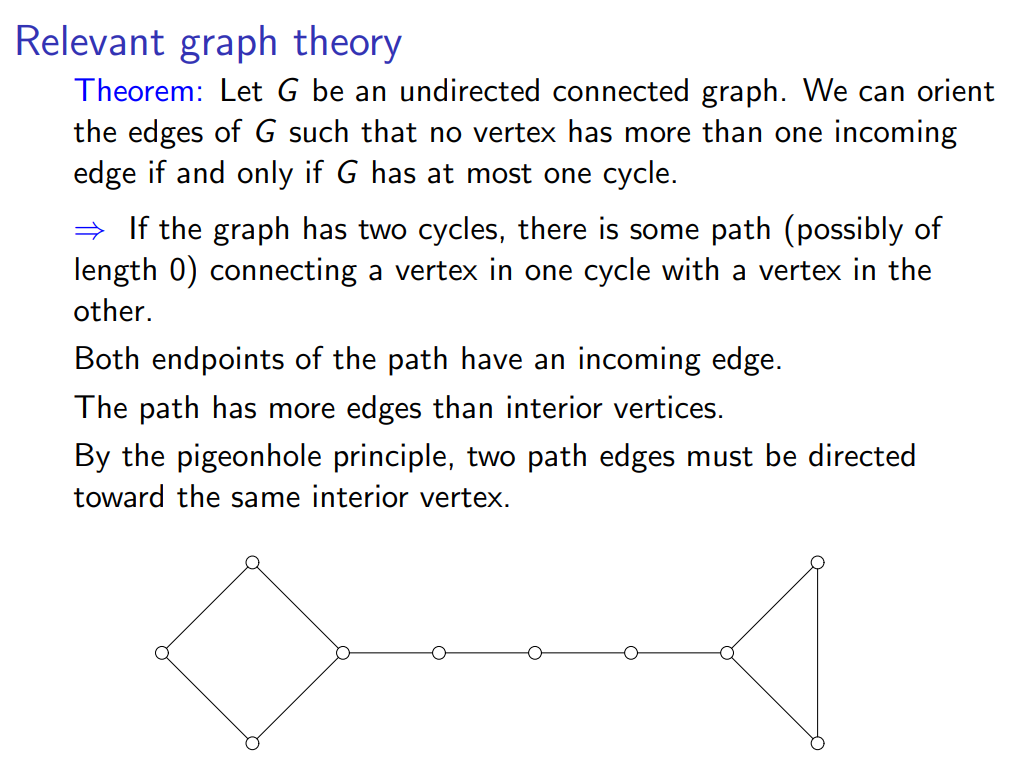
* guaranteed O(1) search
* slower and more complex set operation

### Cuckoo hashing (Graph)

undirected graph, good insertion (*k*) iff the connected component (*k path*) contains at most one cycle

directed graph, valid vertex only has at most one incoming edge ⇐⇒ G has at most one cycle

⇒ impossible to have three paths to one vertex

…

Two conditions

no cycle ⇒ tree

one cycle ⇒ orient all cycle edge around cycle, direct non-cycle away from the cycle

* when inserting to cuckoo hashing, how far can we go until we encounter the two cycles

Proofs… N vertices. When the number of edges ≤ · N for small

expected number of components containing a cycle =

E[number of vertices in 1-cycle components] = O(1)

expected time for the sequence of operations = O(1)

Conclusion

search = O(1) worst-case time

any sequence of n operations = O(1) expected time per operation

### Hash Functions

Def

From K possible keys to N possible index values

All functions are equally likely

k-dependent hash function

k is a small integer (its value ⇐ algorithm)

k-independent ⇒ for every k-tuple of keys, all k-tuples of indices are equally likely

if we have k + 1 keys, they are not promised to be independent

* Hash chaining needs 2-independent for expected time to be O(1)
* Linear probing needs 5-independent…
* Cuckoo hashing needs -independent

Generate k-independent

choose a prime number p >> N

randomly choose k numbers () in range of [0, p-1]

h(x) = (() mod p) mod N

* constant time for fixed k. BUT slow ⇐ multiplication is slow

Tabulation hashing (assume 32-bit)

Preprocessing

build table of 256 random numbers each

Compute

partition k into 4 bytes →

return ^ ^ ^

Properties

3-independent

constant expected time per operation for hash chaining, linear probing, cuckoo hashing

## Heap

# heap in python. Sorted by value

Ex. Dijkstra’s Algorithm to find shortest paths from a start vertex s

Heap Sort

Basic Version

make S into a priority queue Q

while Q is not empty:

find and remove the most urgent item x

output x

In-Place Version

make S into priority queue Q prioritized by value

while Q is not empty:

find and remove the most urgent item x

put x to start/end of the list

### Binary Heap

O(log n) time for all operations except making a heap from a set of n items (O(n) time).

for each non-root node x:

priority(parent(x)) <= priority(x)

implicit complete binary heap

Root = A[0]

Children(A[i]) = (A[2i + 1], A[2i + 2])

Parent(A[i]) = A[floor( (i−1)/2 )]

Heap Operations

find min = O(1)

delete min

move item in A[n-1] to A[0]

fix heap by siftDown or siftUp

fix heap

def siftDown(i):

while A[i] has child with better priority:

sweep A[i] and the better child A[j]

i = j

def siftUp(i):

while i >0 and A[i] is better than its parent A[j]:

sweep A[i] and A[j]

i = j

* worst-case time for siftDown and siftUp =
* if we have k children, siftDown will be more expensive than siftUp

delete at location i

def delete(i):

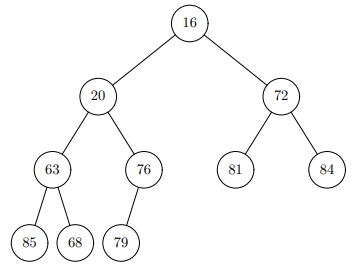
A[i] = A[n-1]

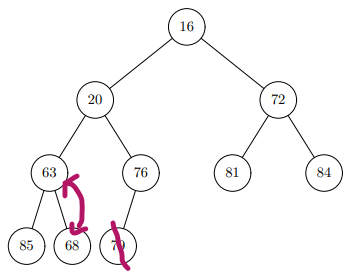
decrement array size

siftUp(i)

siftDown(i)

Ex. delete(63) → 63 becomes 79 → 79 is sweeped with 68



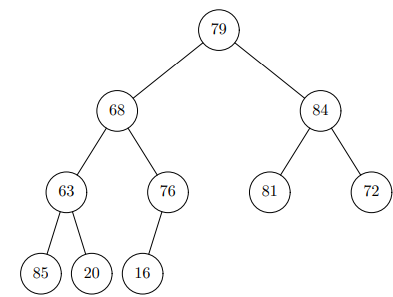


make a heap

def makeheap():

for i in range(n-1, -1, -1):

siftDown(i)



16

20 72

63 68 81 84

85 79 76

* if you are at the bottom, siftDown is less expensive. If you are at the top, siftUp is less expensive

Analysis of makeheap

after iteration i, tree rooted at node i is heap-ordered

at height j

O(j) comparisons to shift node down

height j is same as depth floor(log n) - j

number of nodes at height j

insert

def insert(x):

increment array size

H[n-1] = x

siftUp(n-1) #move up until parent < x

In-Place Heapsort

Make S into a priority queue Q prioritized by -value

//Larger items are more urgent

while Q is nonempty:

find and remove the most urgent item x

put x at front of ‘‘largest items’’ list

// this is location freed up as heap shrinks by one

…

→ Dijkstra’s

while Q is nonempty:

find and remove most urgent vertex v

for each edge v → w:

D[w] = min(D[w], D[v] + length(v→ w))

### K-ary Heap (Improved Dijkstra’s)

Many changeKey and fewer deleteMin) → from binary to k-ary heap

Operations

def siftUp(i): // k-ary heap (same as binary heap)

while i > 0 and A[i] has better priority than its parent:

swap A[i], parent

i = index of parent

def siftDown(i): // k-ary heap (different from binary heap)

j = location of child of A[i] with best priority

while j is in range and A[j] has better priority than A[i]:

swap A[i], A[j]

i = j

j = location of child of A[i] with best priority

* height is smaller when k > 2
* siftUP = O(height)
* siftDown = O(k\*height)

Analysis of operation

height =

deleteMin

k comparisons at each level

time = ⇒ slower

changeKey

1 comparison at each level

time = ⇒ faster

total time =

best value of k happens when

⇒ k = m/n

with best k, total time =

assume for c > 1 then we have = O(m)

Summary

Binary heaps:

deleteMin in O(log n) time

changeKey in O(log n) time

Dijkstra’s algorithm in O(m log n + n log n) time.

k-ary heaps:

deleteMin in

changeKey in time

Dijkstra’s algorithm in O(m) time when for c > 1

### Fibonacci Heaps

Advantages

O(1) amortized Decrease Priority operation

O(log n) amortized Delete Min operation

Dijkstra’s algorithm in O(m + n log n)

Fibonacci Heap Node contains

data, priority, boolean = True if … and …

v is not root

1…\* child of v was removed after v most recently became a non-root (after v becomes child of m, v lost the second child)

parent pointer,

pointer to left and right sibling (circular doubly linked list)

number of children, pointer to one child,

Potential Function

Easy Operations

Find Min:

Return v pointered to by best root pointer

Actual time O(1), ∆Φ = 0, amortized time is O(1).

Make (a new Fibonacci Heap from n items):

Create a new root node for each item, link them together.

Actual time O(n), ∆Φ = n, amortized time is O(n).

Insert:

Create a root node for the item

Compare with previous best root, update if necessary

Actual time O(1), ∆Φ = 1, amortized time is O(1).

Merge (two Fibonacci Heaps):

Concatenate root lists by changing pointers of the double-linked list

Compare best roots to determine best root of merged heap

Actual time O(1), ∆Φ = 0, amortized time is O(1)

Decrease Priority Operation

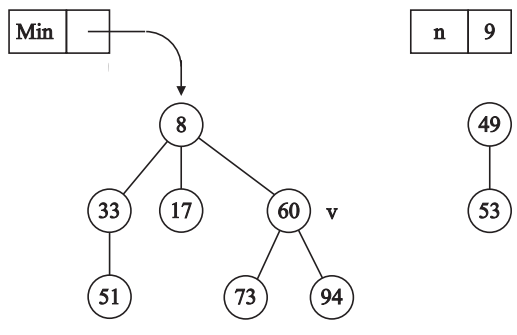
x is root ⇒ decrease its priority (we might need to check the min flag?)

x is not root ⇒ make it a root node first. x’s parent = p

if p is not root and flag is False ⇒ set it to True

if p is not root and flag is True ⇒ make p the root (we then look at p’s parent…)

Ex. decrease priority (v, 5)

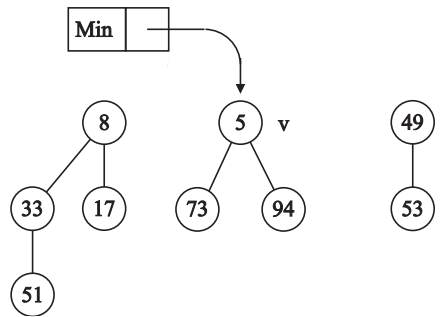


take v off the sibling list

add v to the sibling list of 8 and 49

root pointer to v

v’s previous parent is root ⇒ done



Pseudocode

def promote(x):

if x is not a root

p = x.parent

remove x from p’s children

remove x from sibling list

add it to root list

x.flag = False

if p.flag:

promote(p)

else if p is not a root:

p.flag = True

def decreasePriority(x):

promote(x)

compare x to best root, change if better

Analysis: = # of promotions

actual time = O(k+1)

+k roots created

-2(k-1) for k-1 flags change True → False

+2 possibly for last flag change False → True

⇒

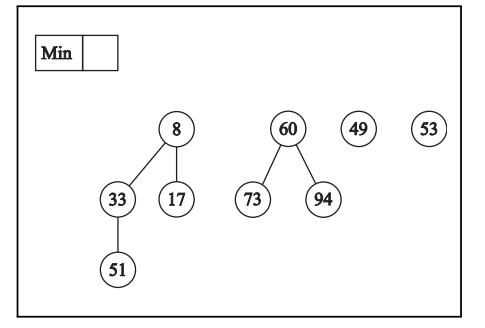
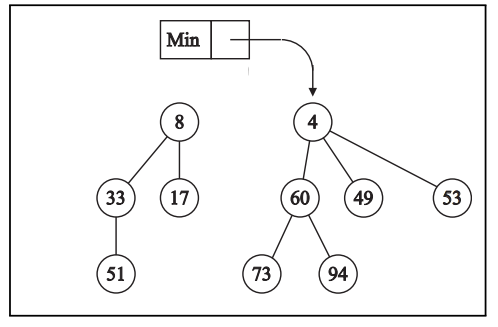
Amortized time = c\*(k+1) + c\*(- k+1) = O(1)

Delete Min Operation

Merge children of deleted node into list of roots

* The boolean flag of each child of deleted node must be set to False

Ex. delete 4



Improve forest structure to decrease its potential ⇒ Force all root nodes to have different numbers of children

M = maximum possible number of children a node can have

M = O(log n)

pseudocode

C = [None] \* (M+1)

R = all tree roots #R is a collection of tree roots not yet in C.

while R is not empty

remove an element x from R

if C[x.childCount] == null:

C[x.childCount] = x

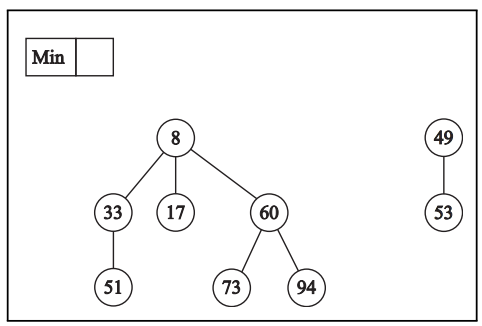
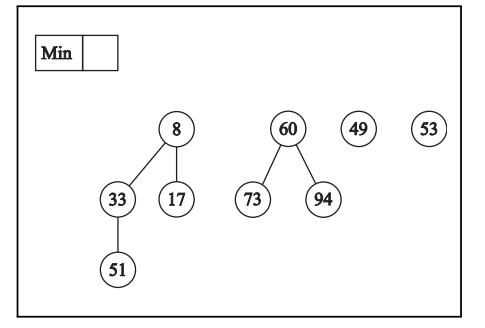
else: #force all root nodes to have different # children

combine x, C[x.childCount] into a new tree y

add y to R

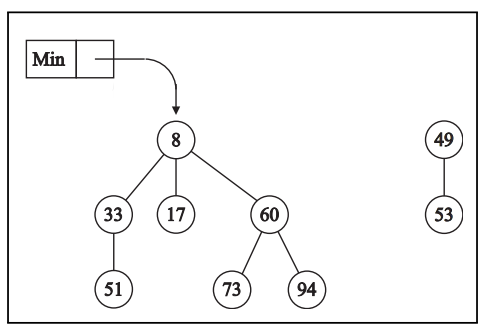
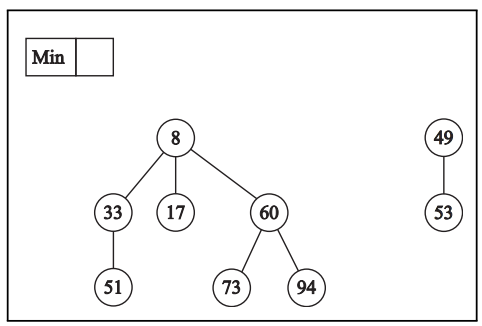
C[x.childCount] = null

Ex. 8 has 2 children, so does 60 ⇒ 60 will be merged to 8 and re-add it again



Find new best root (by traversing new root list sequentially)

Ex.



Analysis of Delete Min

Merge step

actual time = O(M)

change in number of tree roots <= M

Improvement step

actual time = O(t)

change in number of tree roots <= M+1 - t <= O(M) - t

Find a new best root step

actual time = O(M)

Overall actual time = O(t) + O(M)

Overall

Overall amortized time = O(M)

M = log n if we can prove

Suppose a node v in a Fibonacci heap has i children. Number v’s children from 1 to i, according to the order they were added.

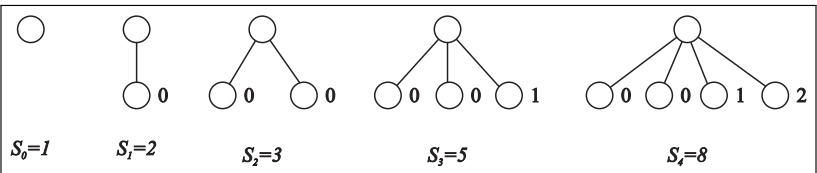
For each j ∈ 2, 3, . . . , i:

When child j became a child of v, it had at least j − 1 children. ( ⇐ Delete Min).

Since then, child j has lost at most one child. (Otherwise, it would be a root ⇐ Decrease Priority)

⇒ Child j has at least j − 2 children, for j = 2, 3, . . . , i

Ex. S(i) be the minimum number of nodes in a tree whose root has i children



S3 has 5 because child 3 has at least j-3 children, which is 1 child

……

Summary

| Fibonacci Heap Operations | Amortized Time | Also worst-case? |
| --- | --- | --- |
| Insert  MakeHeap  FindMin  Merge  Decrease Priority  Delete Min | O(1) O(n)  O(1)  O(1)  O(1)  O(log n) | Y  Y  Y  Y  N N |

| Dijkstra’s Algorithm Operation | Count | Time | Total Time |
| --- | --- | --- | --- |
| MakeHeap  Delete Min  Decrease Priority | 1  O(n)  O(m) | O(n) O(log n)  O(1) | O(n)  O(n log n)  O(m) |
| Total |  |  | O(m + n log n) |

## Bloom filter

### Set

Bitmap O(1+)

01001010 for x y z

{x,y,z} ⇒ (1<<x) | (1<<y) | (1<<z)

union of S, T ⇒ S | T

intersection ⇒ S & T

⇒ (1<<x) & S != 0

add x to S ⇒ S = S | (1<<x)

remove x from S ⇒ S = S & ~(1<<x)

Hash Table (python) without values

add/remove O(1)

build a set O(n)

union O(n)

def union(S,T):

make new set X

for x in S: X.add(x)

for x in T: X.add(x) –no duplicate

return X

intersect O(n)

def intersect(S,T):

make new set X

for each x in smaller of S,T:

if x is in other set:

X.add(x)

return X

check if x exist O(1) for hash table

test equality of two sets O(n)

check if size is same → loop over all elements

### Bloom Filter

Hash Table H of N > n\*k bits

Hash function maps each element to k-tuple of bits in H

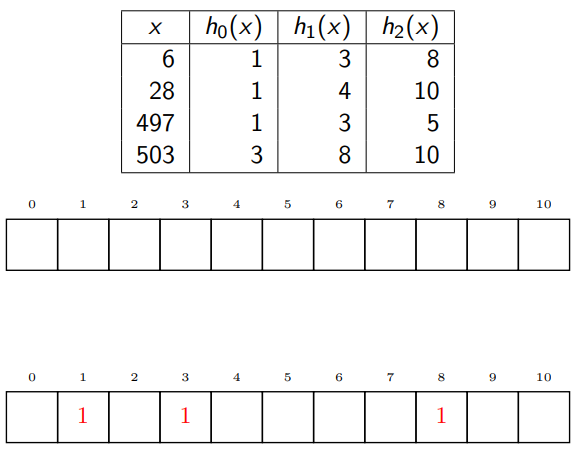
initial state: all bits = 0

add(x): hash x, set all k bits to 1

check if x exist: hash x → test whether all k bits are 1

union(S,T): S | T

Ex. insert 6 gets the following. testing 503 before insertion will return True



Application

firewall with small “white list” (might allow unauthorized user to go through)

“black list” prevents much good traffic but absolutely denies all bad traffic

Advantages and Disadvantages

Fast but Inexact large sets

False positive (check if exist → yes might be a lie) are possible

False negative are not possible (not exist → not exist for sure)

⇒ True always

⇒ True/False

supports adds, union, BUT doesn’t support efficient intersection, removal

False Positive Rate (k, n, N)

p\*N 1-bits and (1–p)\*N 0-bits where p = probability that a bit is 1

for a key x is not in the set but reported to be:

is the easy bound but accurate when n\*k <<N (more empty)

Better bound

assume all key hash values are independent

after n insertions

where

nk/N is constant ⇒ false positive decrease exponentially with k

Counting Bloom Filter

each cell store # of set items mapped to it

remove element subtract 1 from counts ⇒ make False Negative possible

storage for counts

worst-case = O(n)

average counts = O(log n/log log n)

O(log log n) bits per cell are sufficient

Cuckoo Filter

Difference between cuckoo hashing

instead of storing item x, we store a fingerprint with a fixed bit length

fingerprint = fp(x) with hash function

each cell stores a fixed number k of fingerprints

one table instead of two tables

check if x exists

check if fp(x) is in the bucket keys in x’s two locations (false +, but not false –)

insert x

store fp(x) and x to the first location of x

if bucket is full, delete randomly a item form k items, insert it to its other location ⇒ problem with unknown value of the deleted item

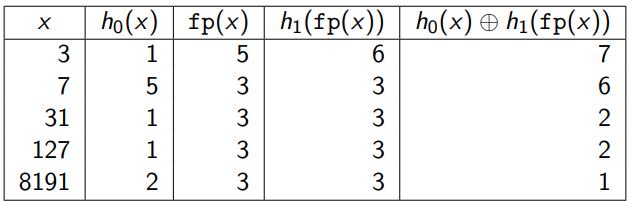
* One alternative = two hash function h0 and h1
  + h0(x) as location 1, h0(x) XOR h1(fp(x)) as location 2
  + one location of x is z ⇒ other location = z XOR h1(fp(x))
* simplified alternative = h0
  + h0(x) as location 1, h0(x) XOR fp(x) as location 2
  + one location of x is z ⇒ other location = z XOR fp(x)

delete x

fp(x) in both x’s locations and remove if found

if not previously added ⇒ False Negative introduced

Ex. fp(3) = 3



Adds 3

0 1 2 3 4 5 6 7

5

Adds 7 (all membership tests are correct)

0 1 2 3 4 5 6 7

5 3

Adds 31 (127 and 8191 are incorrect)

0 1 2 3 4 5 6 7

(5,3) 3

Adds 127 (8191 is incorrect) ⇒ 5 for 3 move from 1 to 7

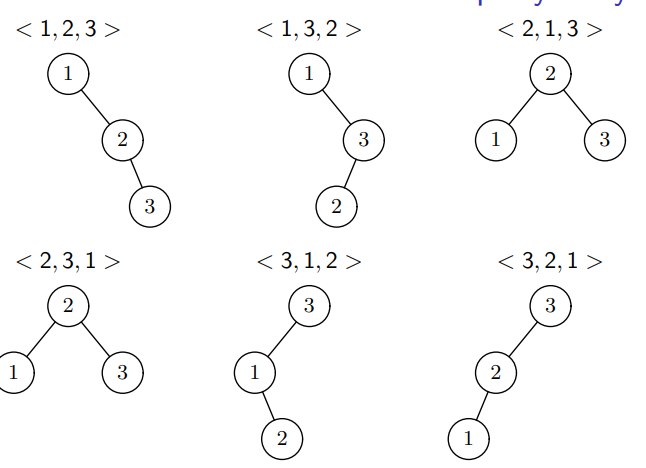
0 1 2 3 4 5 6 7

(3,3) 3 5

## Binary (Search) Tree

Tree Insertion in Random Order (insert form a fixed set)

Ex. nice tree (balanced) is more likely



Assumption

keys are 0, 1, 2, …, n-1 when we insert x

Expected cost of insertion x = O(log n) without deletion if random insertion

Not random insertion ⇒ maintain tree height to be (log n)

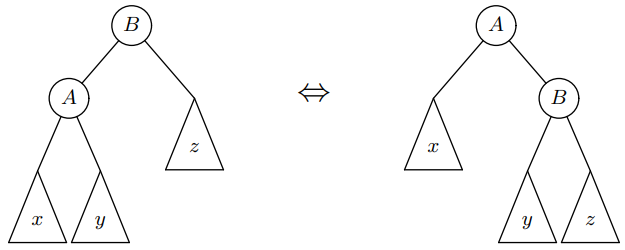
Rebuild Entire Tree

Balance by Rotation

Ex. ⇒ right rotation, ⇐ left rotation

Right rotate on v: left child of v becomes v’s parent, v becomes its right child

Left rotate on v: right child of v becomes v’s parent, v becomes its left child



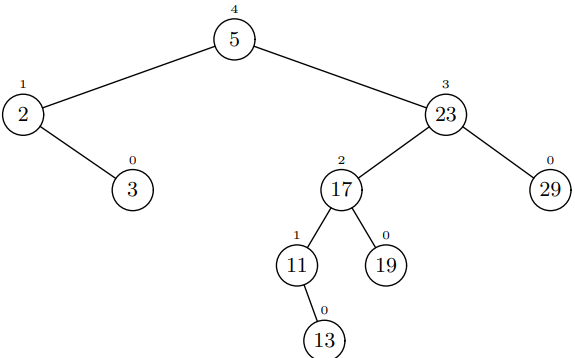
### AVL Tree

(difference between height(left subtree) and height(right subtree) <= 1

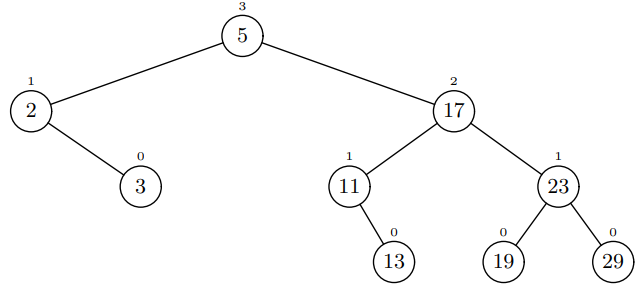
insertion:

standard insert

Ex. insert 13



right rotation on 23



Deletion **O(log n) rotations**

### Red Black Tree

Rules

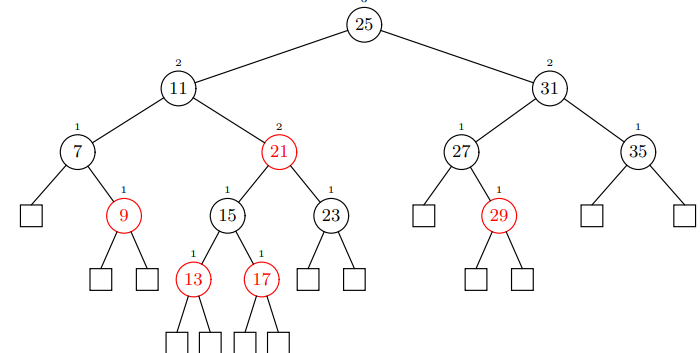
Every node is colored either red or black

All external nodes are black

The children of a red node must be black

The path from every external node to the root contains the same # black nodes

The black height of x= # black nodes on a path from x to a leaf, not counting x



Insertion

standard insertion.

New node replaces an external node

New node is colored red with two (black) external children

All properties are satisfied except that the parent of new (red) nodes may also be red.

Red-red anomaly can be fixed with

Recoloring of at most O(log n) nodes

At most 2 rotations.

Deletion **O(1) rotations**

Rank-Balanced Trees

Rules

rank != height(subtree)

(rd)rank difference = rank(parent(node)) – rank(node)

height(node) = O(rank(node))

rank(node) = O(log (size of subtree rooted at node))

Special Cases

AVL ⇐ rank= height, rd = 1 or 2,

Red black tree ⇐ rank = black height, rd = 0 or 1

WAVL (wake avl) rd = 1 or 2, update needs 2 rotation

### WAVL Tree

Rules

rank ~ height

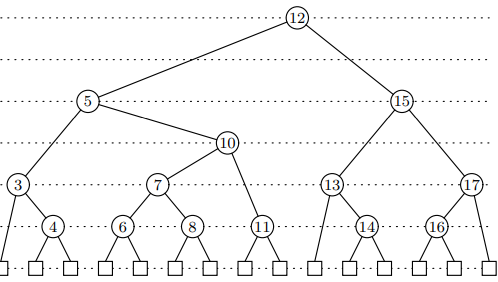
rd(node) = rank(parent(node)) - rank(node)

Ex. Different types of node

(1,2) node = (5), (10), (13)

(1,1) node is (4)

(2,2) node = (12), (15)



Properties that needs to be satisfied

rd(any non-root node) = 1 or 2. (level diff = 1 or 2)

External-Node rank = 0 (square block)

Internal-Node with two external children (1,1) node and has rank = 1??

Relation

every AVL tree is a WAVL tree (not converse)

every WAVL tree can be a red-black tree (not converse)

Insertion

Search for the value

Arrives at an external node (with rank 0), replace the external node by an internal node x, containing the value to be inserted.

The new internal node x will have:

rank 1

Two external children at rank 0.

In the special case where the tree was empty, x is now the root.

Otherwise

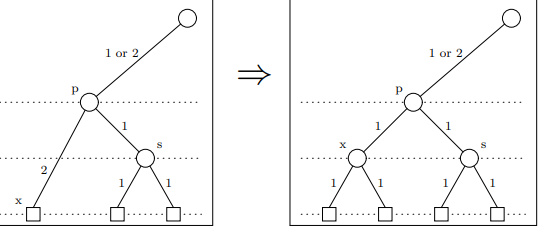
Let p be the parent of x

replace x by rank 1

node with two external children

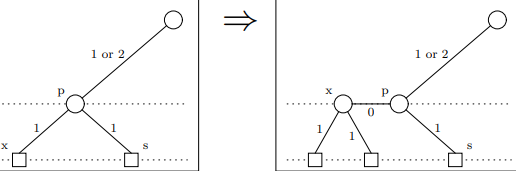
if p has rank 2:

done



else:

violation\_fix



Violation Fix

x is a node with rank-difference 0

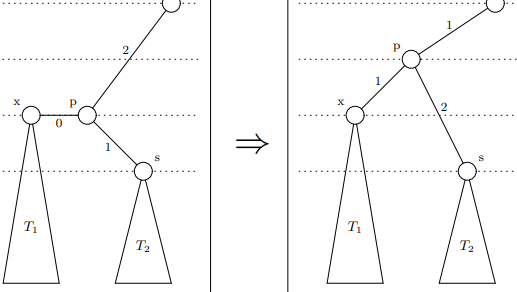
p is the parent of x

s is the sibling of x

We assume that x is the left child of p. (Otherwise, mirror image).

if rd(s) == 1: #rd(x) = 0 and rd(s) = 1

promote p



//if p is root: done

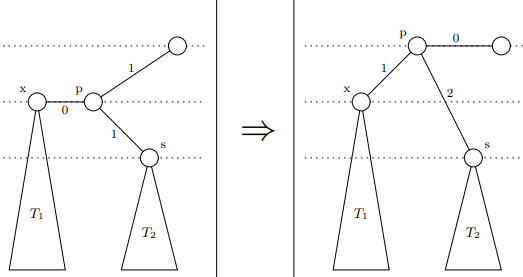
else: #rd(s) = 2

#violate at p

x = p

p = p.parent

continue



Violate fix of rd = 2

x is a node with rank-difference 0

p is the parent of x

s is the sibling of x, rd(s) = 2

We assume that x is the left child of p

t = child(x) with rd(t) = 1

if t is left child:

rotate right at p

p.rank -= 1

else:

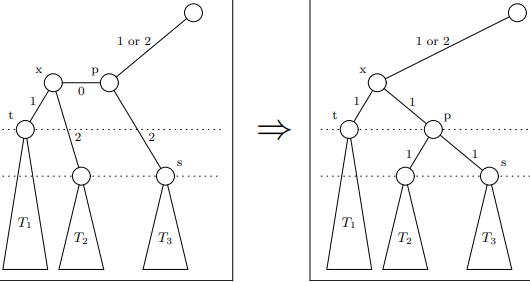
rotate left at x

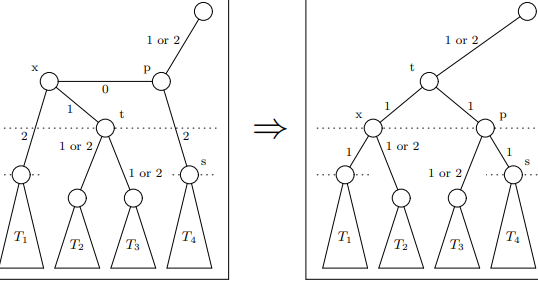
rotate right at p

p.rank -= 1

x.rank -= 1

t.rank += 1





Deletion

Known

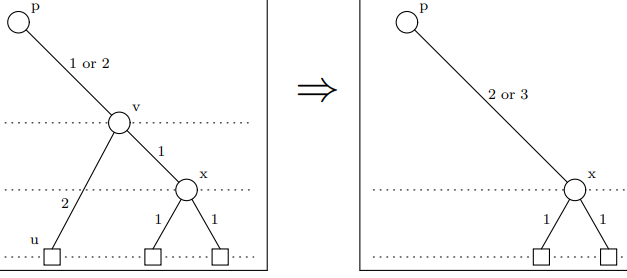
Let v be the node that is actually deleted.

v has at least one external child, u.

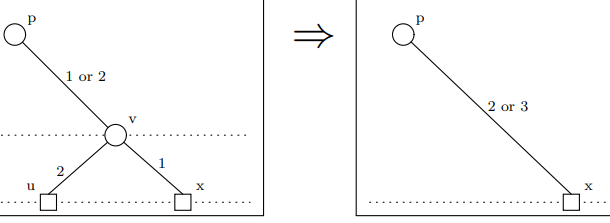
p = parent of v

x = other child of v

If x is an internal node:

x has rank 1, v has rank 2, p is null or has rank 3 or 4

If x is an external node:

x has rank 0, v has rank 1, p is null or has rank 2 or 3

Operations:

replace v with x

remove u

if v is root: done

if v had rd == 1: done

else: v had rd =2 and x has rd = 3 (violate on x)

rd 3 violation

rd(x) = 3

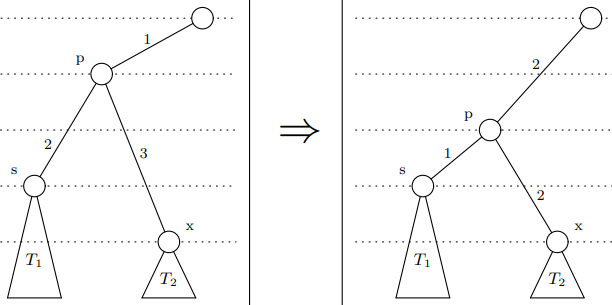
p = parent(x)

s = sibling(x)

//assume x is the right child

if rd(s) == 2: //push it up a level

demote p // p down a level



if p had rd 1:done

elif p is root: done

else:

violate at p

x = p

p = p.parent

else: //rd(s) == 1 two sub cases

if rd(s.children) = (2,2)

demote both p and s

if p had rd 1:done

elif p is root: done

else:

violate at p

x = p

p = p.parent

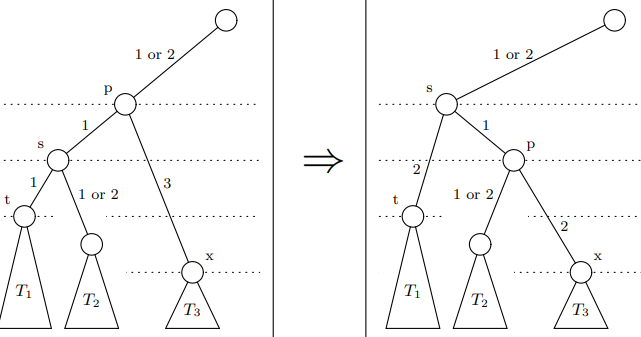
else:

if rd(s.left) = 1:

rotate right at s

p.rank –=1

s.rank += 1



else:

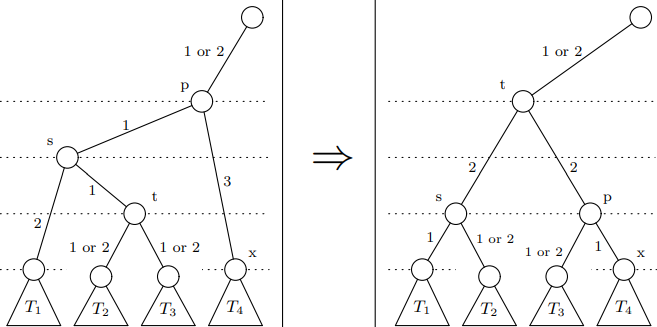
rotate left at s

rotate right at p

p.rank –=2

s.rank –=1

t.rank += 2

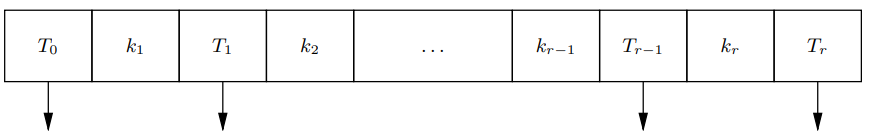


Performance of insertion + deletion: WAVL > Red Black Tree > AVL Tree

## Balanced Tree

Multiway tree

# keys = # children - 1.



(a,b)-tree multiway tree

b = 2a – 1 and every node has [a, b] children

root has at least 2 children. non-root node has at least 1 child

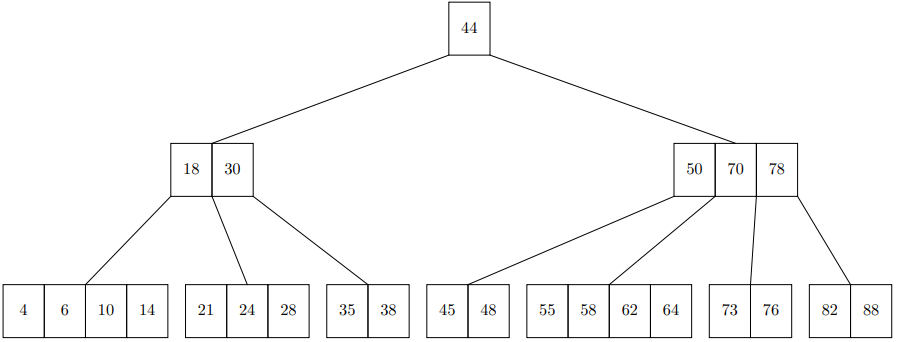
all leaf nodes are at the same level

### B-Tree: allow variable number of keys per node

[m/2, m] children and m is odd for this lecture

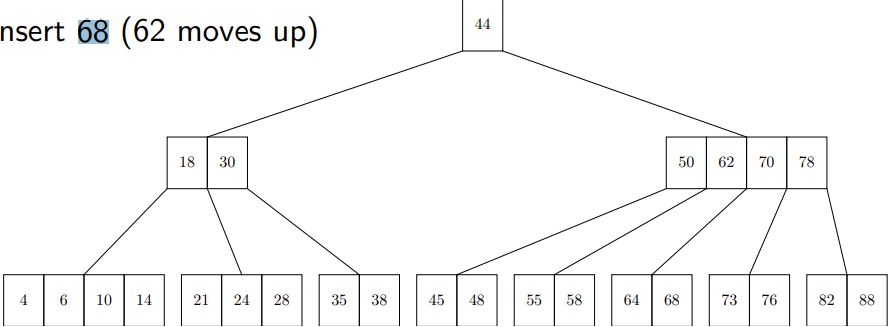
root has at least 2 children

Ex. (3,5) tree. B-tree of order = 5



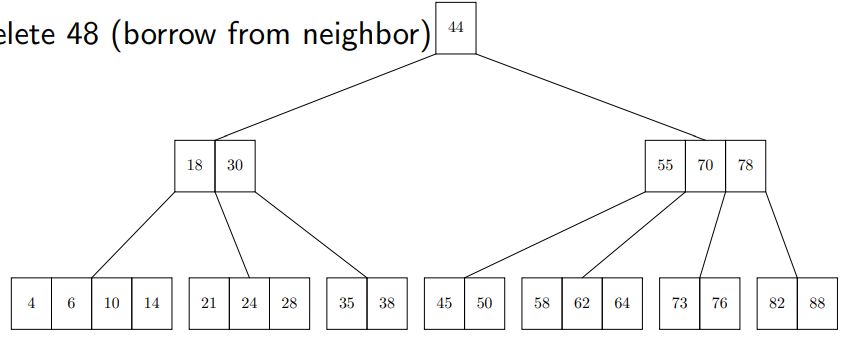
Insertion on overflow ⇒ split into two children → push up the leftmost one to parent

Ex. insert 68 → split → push 62 up

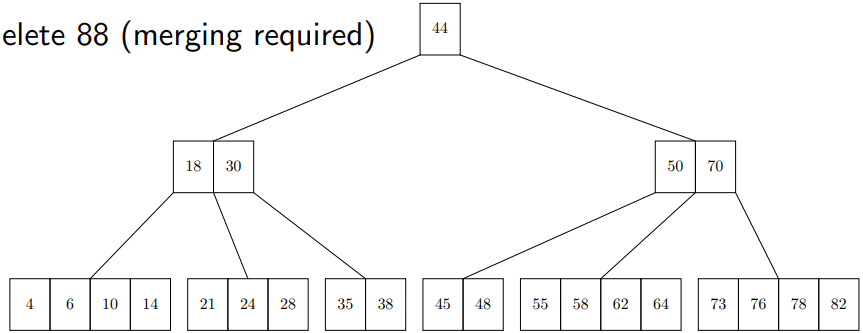


Deletion on underflow ⇒ borrow underflow ⇒ merge → pull the key in parent down (delete)

Ex. delete 48 → borrow from parent (50) and parent borrow from next children (55)



Ex. delete 88 → 78 is pulled down to the child and merge two children



Performance

all operations access nodes (n = # keys)

### Treaps = Tree + Heaps

Basics rotate based on priority of the inserted/deleted node

node = key value + child pointer + priority value (unique and not change)

def insert(self, k): #rotate up until the priority is correct

v = Insert k using standard binary tree insertion

# v is a leaf node

v.priority = random()

while (v is not the root and v.priority > v.parent.priority):

if v is a left child:

rotate\_right(v.parent)

else:

rotate\_left(v.parent)

def delete(self,v): #rotate child with higher priority

while v is not a leaf:

if left(v) == None:

rotate\_left(v)

elif (right(v) == None or v.left.priority > v.right.priority):

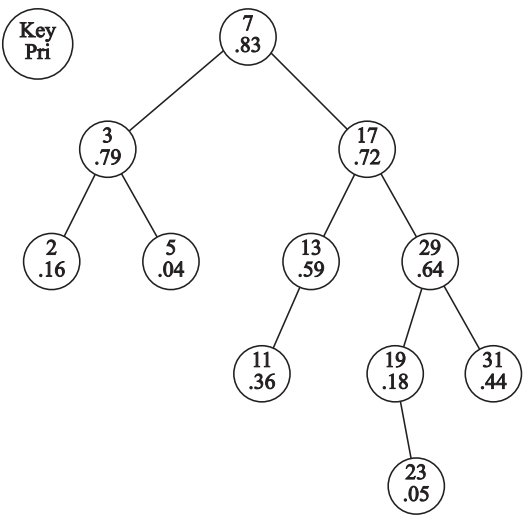
rotate\_right(v)

else:

rotate\_left(v)

Delete the leaf node v

Ex. delete 29 → 31 is up



### Zip Trees

Skip List Basics

list of level = sublist of level down

random decision if go up one more level ⇒ tall node might be a problem

O(log n) expected search,insert, delete.

O(n) expected space

Zip Trees similar to Treaps but diff ranks

The rank is exactly like the height of a node in a skip list:

0 with probability 1/2

1 with probability 1/4

2 with probability ⅛

Rank Rule

v.rank > v.left.rank if v.left is not null.

v.rank >= v.right.rank if v.right is not null.

if v.rank = w.rank and v.key < w.key ⇒ w must be in subtree at v.right and v cannot be subtree at w.left

Insertion

search path for x.key until we encounter the node y that node x will replace. This will be the first node y such that either:

y.rank < x.rank, or

y.rank = x.rank and y.key > x.key

P = left path and Q = right path

P’s increasing key and non-decreasing rank

Q non-increasing key and decreasing rank

After insertion

if P == null, P =

def unzip(x,y): #insert x to y (to be replaced)

def unzip lookup(k,node):

if node is None:

return (None,None)

if node.key < k:

(P,Q) = unzip lookup(k,node.right)

node.right = P

return(node, Q)

else: // node.key > k

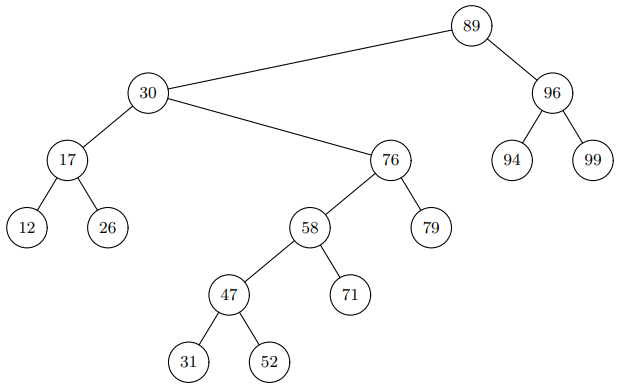
(P,Q) = unzip lookup(k,node.left)

node.left = Q

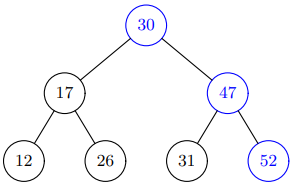
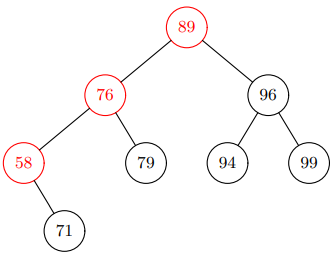
return(P, node)

return unzip lookup(x.key,y)

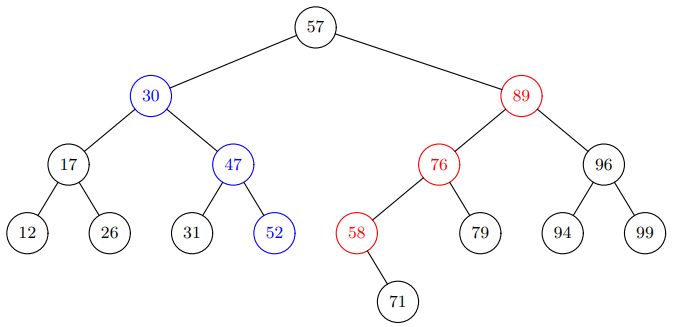
Ex. insert 57



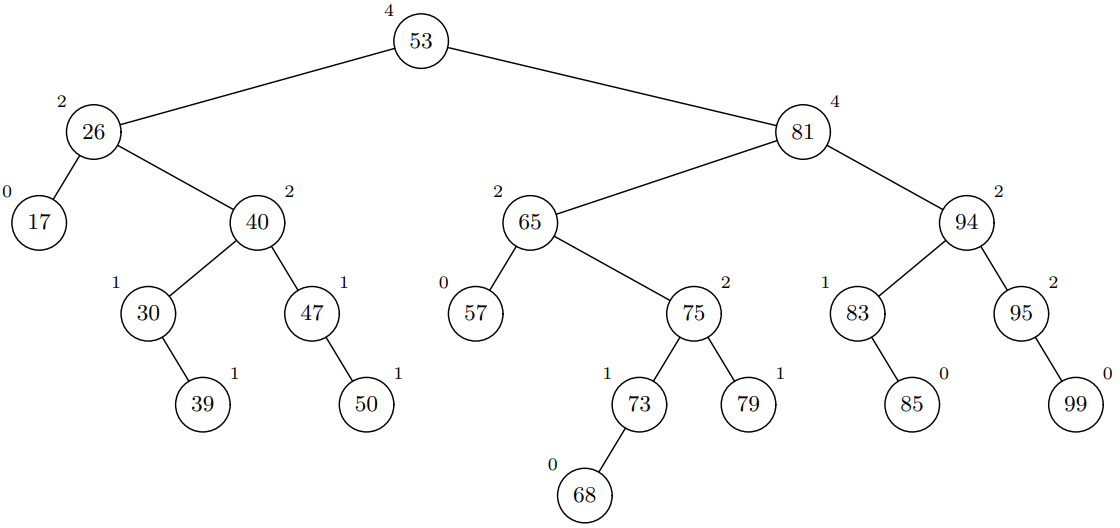
Unzip lookup path of 57 (**89**, 30, **76**, **58**, 47, 52) and split into 2 root children

Make 57 root

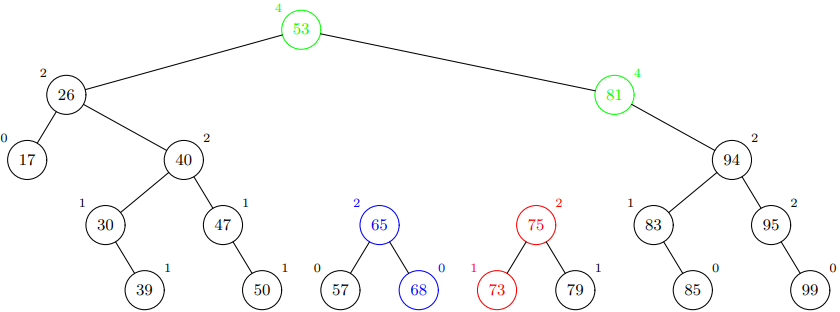


Ex. insert 70 (rank = 3)



Lookup path = 53, 81 65, **75**, **73**, 68

Unzip on 65



Deletion

Search path of deleted node x (left and right)

Merge the two path based on (rank, value)

def zip(x):

def zipup(P,Q):

if P is None: return Q

if Q is None: return P

if Q.rank > P.rank:

Q.left = zipup(P,Q.left)

return Q

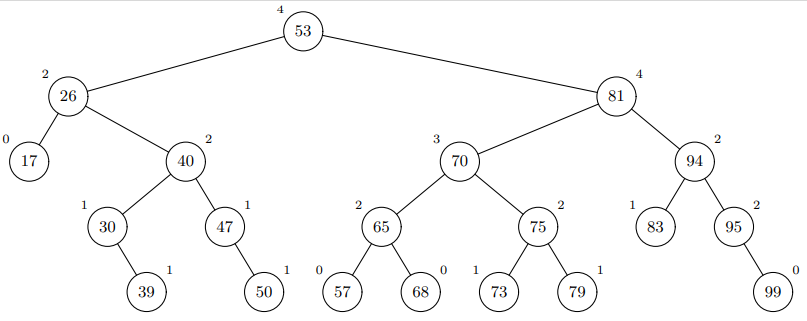
else:

P.right = zipup(P.right,Q)

return P

return zipup(x.left,x.right)

Ex. delete 81



left path 70, 75, 79

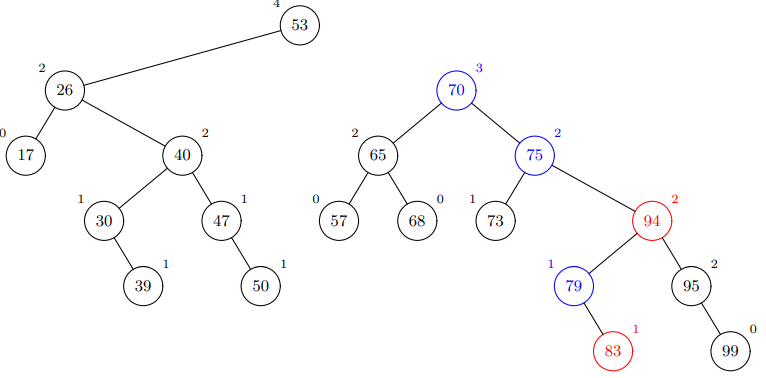
right path 94, 83

70 and 94 ⇒ 74 for higher rank

75 and 94 has the same rank ⇒ 75 for smaller value

94 and 79 ⇒ 94 for higher rank

79 and 83 ⇒ 79 for smaller value



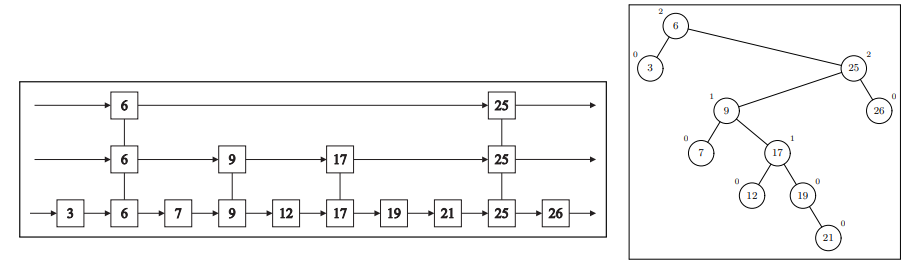
Zip tree vs. Skip list

rank(zip\_tree) = height(skip\_list)

following right of zip\_tree = moving to the right in skip list

following left of zip\_tree = dropping down a level

zip\_tree saves some comparisons and storage



Zip tree vs Treaps

both are binary tree and max-heap on rank/priority

treaps use rotations, zip\_tree uses zip/unzip

treap’s priority

uniform distribution

unique

O(log n) bits needed to represent priority

zip\_tree’s rank

geometric distribution

O(log log n) bits needed to represent rank

same performance bounds on average case rather than worst-case

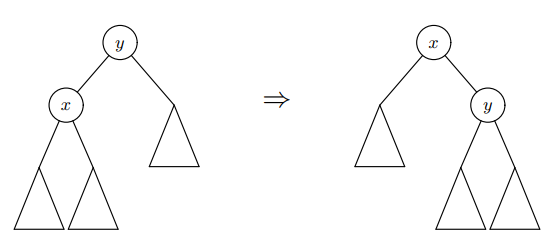
### Splay Trees

Basics

splay at node x ⇒ make x the root

Case 1 - x has no grandparent (zig)

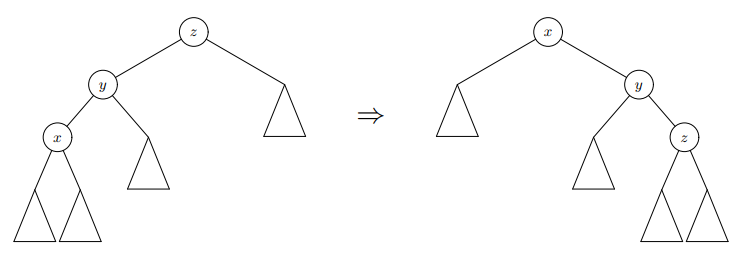
rotate right on x’s parent y



Case 2 - x is LL or RR grandchild (zig-zig) LL ⇒ RR rotations

rotate\_right(x’s grandparent)

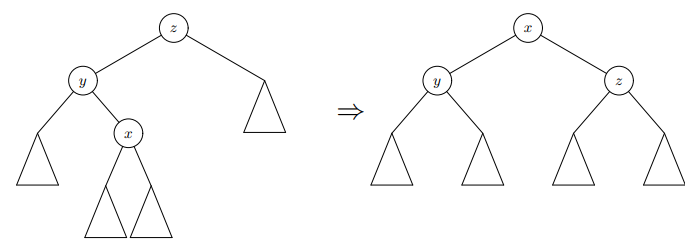
rotate\_right(x’s parent)



Case 3 - x is LR or RL grandchild (zig-zag) RL ⇒ LR rotations

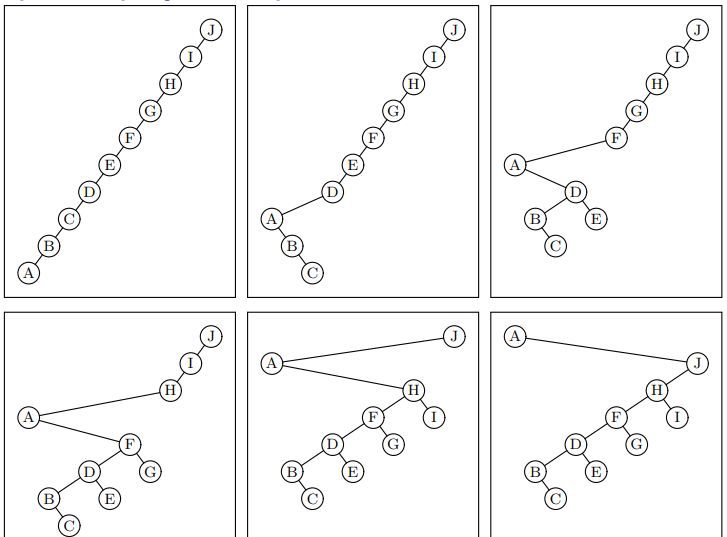
rotate\_left(x’s parent y)

rotate\_right(x’s original grandparent z)

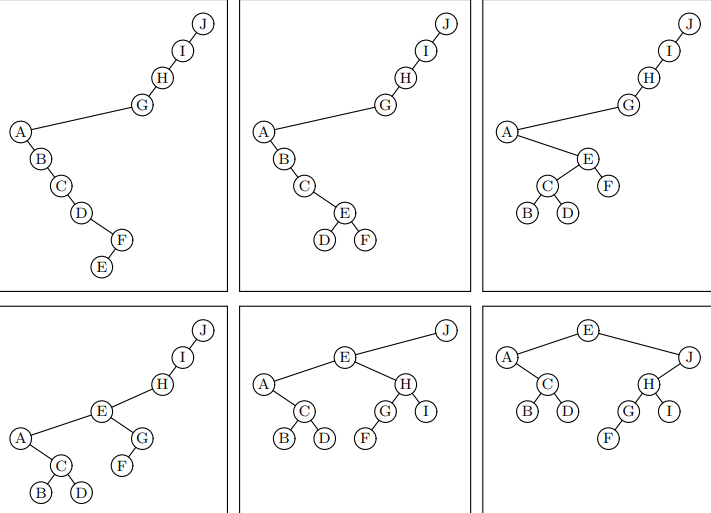


Ex.

Splay on A



Splay on E



Operation **all O(log n) amortized time**

Access(i)

success ⇒ splay on node i

unsuccess ⇒ splay last node on search path

Insert(i)

splay at newly inserted leaf node i

Delete(i)

found ⇒ splay the parent of the deleted node

not found ⇒ splay the last node on search path

Amortized analysis

|*w*| = # external nodes descending from node *w* (# of internal nodes + 1)

(proof not in test)

Online Algorithm = make decision based on current knowledge

Competitive ratio = max(cost of A / cost of optimal choice) #max = worst-case

Ex. Cache management with cache size = k. LRU competitive ratio = k

Ex. maintain a BST for a fixed set of n items

cost(access) = #nodes on search path + #rotations

Dynamic Optimality Conjecture competitive ratio of splay tree = O(1)

## String Matching

Dictionary/Successor problem on strings: Given a word W and alist L of words

Does W appear in L?

If not:

What is the next word after W that appears in L?

How can W be completed to one of the words in L?

Searching word in a large doc

Build a suffix tree

alphabet = word

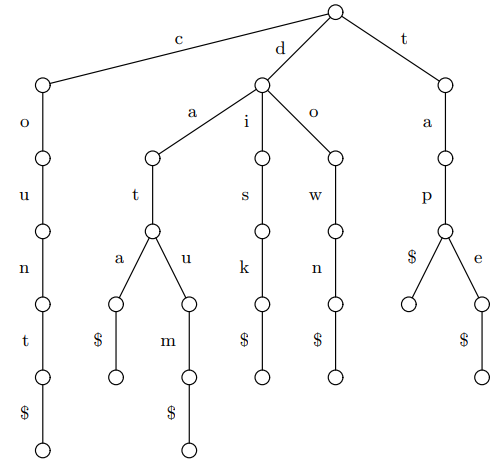
space = O(# words)

leaf node = document ID

### Trie (Prefix Tree)

End-Mark Trie

Ex. S = {“count”, “data”, “datum”, “disk”, “down”}



Implementation Options of the Dictionary

sorted array of (character, node) pair space-efficient but slow search if large amount of pairs

hash table fast but can’t find successors

array indexed by char only good for small alphabet

Operations

def search(q):

node = root

for character c in q$:

if node.child[c] exists:

node = node.child[c]

else:

raise not found exception

(or, for successor)

back up tree until find a node with a next child follow leftmost path down to a leaf

def insert(q):

node = root

for character c in q$:

if node.child[c] exists: node = node.child[c]

else: #make a new path

node.child[c] = new Node(c)

node = node.child[c]

Analysis

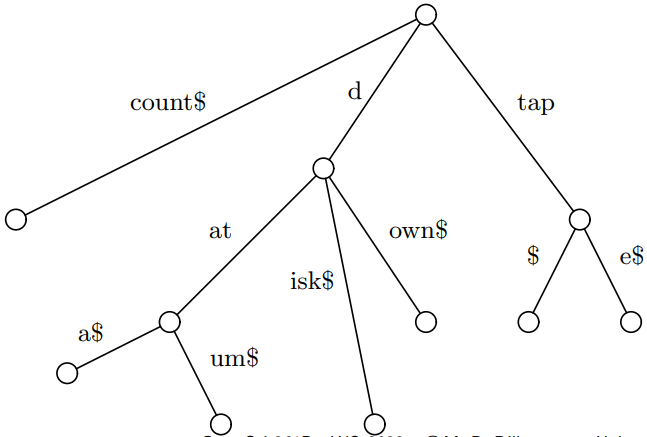
query time = O(|q|) lookups

space: #nodes = O(sum of lengths of all the stored words) = O(input size)

preprocessing = O(input size)

Compressed/Patricia Trie

Ex. S = {“count”, “data”, “datum”, “disk”, “down”, “tap”, “tape”}



Operations

def search(q):

node = root; i = 0; j = 0 #i for incoming string

repeat

if i == len(node.incoming string)

if q[j] == ’$’:

return FOUND

i = 0

node = node.child[q[j]]

else if node.incoming string[i] == q[j]:

i = i + 1; j = j + 1

else:

q is not in dictionary

### Suffix Tree

End-Mark Suffix Tree

Def. In other words, the suffix tree for S is the trie for these n + 1 strings with end markers:

I S[0]S[1] . . . S[n − 1]$

I S[1]S[2] . . . S[n − 1]$

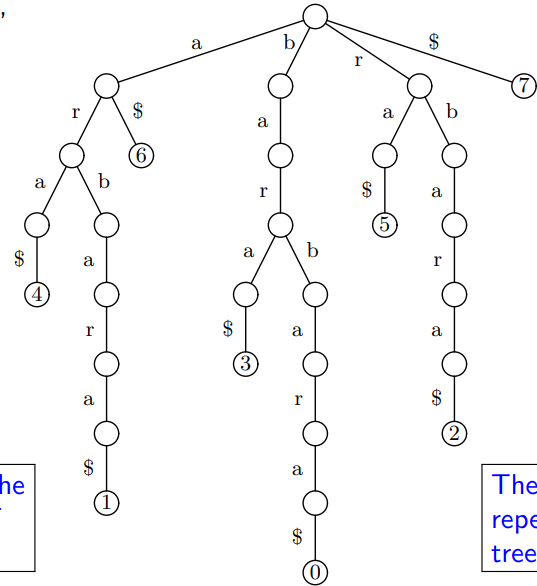
I . . .

I S[n − 2]S[n − 1]$

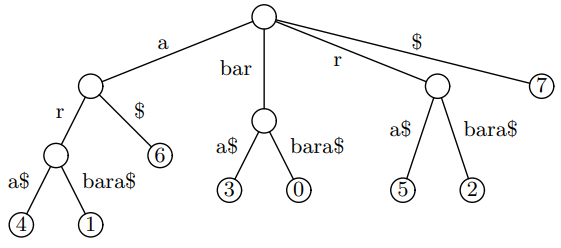
I S[n − 1]$

I $

Ex. barbara (leaf node is the starting point of the suffix)



Ex. Compressed Suffix tree

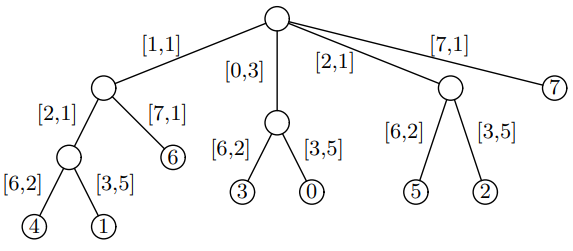


no need to reach the $ to test if q is a substring

If we reach the end of q without finding a mismatch:

Locations of input matching q are given by leaves descending from the current position in the tree

Ex. Compacted Suffix Tree. Store [starting position, length] in the node



## Range Queries

A range query is decomposable if there is some binary operation ⊗ such that:

⊗ is associative and has an identity operand

If Q is a region and we split Q into two disjoint regions X and Y :

query(Q) = query(X) ⊗ query(Y )

Ex. decomposable query

count: ⊗ = integer addition

sum: ⊗ = integer addition

listing: ⊗ = union

most urgent point : ⊗ = minimum

Ex. non-decomposable query

average: avg(X) ⊗ avg(Y) != avg(Q)

(sum, count) is decomposable to replace average

Dynamic 1-dimensional Decomposable Range queries

Data has keys that are real numbers. (Data may also have other associated values)

Range: closed interval [L, R] (i.e., L ≤ key ≤ R.

Dynamic: data points can be added/removed

Ex. Sum Range Queries

item = (key, value) where key = real # and value = integers

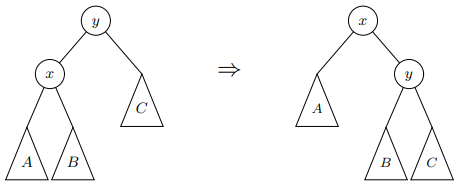
query(L, R) =

items in a balanced BST on item.key

each node x has a mutable value = sum(subtree at x)

insert/delete x ⇒ add/subtract x.value on its search path

rotation on y



sum(x)= sum(y) and recompute sum(y)

range query

find the first node M whose key in the range

find the left and right bound of the range in M.left and M.right

Ex. find query [10, 87]

root is the first node in the range

res = 79

lookup(10) in root.left

M (19.1, 11) in range

res += 11 + aggregate(M.right) #11 + 35

search M.left

M (5.6, 32) not in range → search M.right

M (15.7, 75) in range

res += 75 + aggregate(M.right) #75 + 0

search M.left

lookup(87) in root.right

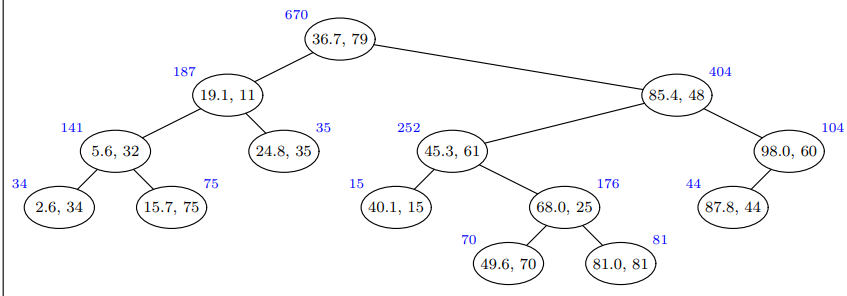
M(85.4, 48) in range

res += 48 + aggregate(M.left) #48 + 252

search M.right

M (98.0, 60) not in range →search M.left

M (87.8, 44) not in range → search M.left



General Case with Count example

Note that augment(root) = value(x) ⊗ augment(x.left) ⊗ augment(x.right)

Query

Perform binary search to find the lowest and highest node v in [L, R]

Left tree of root:

x.key < L ⇒ search x.right

x.key ≥ L:

add augment(x.right)

search x.left

def query(T,L,R):

if T.key < L:

return query(T.right,L,R)

if T.key > R:

return query(T.left,L,R)

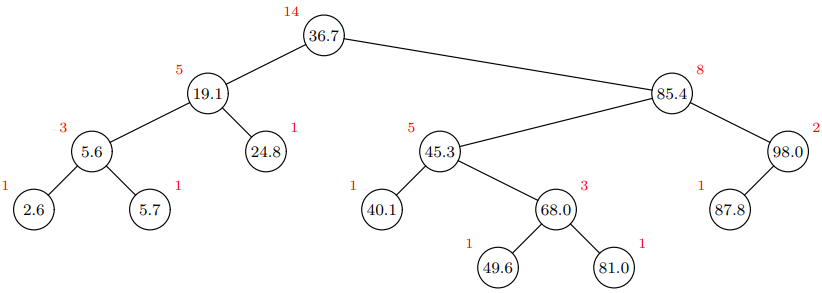
if R == +∞:

return query(T.left,L,R) ⊗ value(T) ⊗ augment(T.right)

if L == -∞:

return augment(T.left) ⊗ value(T) ⊗ query(T.right,L,R)

return query(T.left,L,+∞) ⊗ value(T) ⊗ query(T.right,−∞,R)



Analysis of 1D Dynamic Range Query

query answer is in O(1) space + ⊗ in O(1) time ⇒ O(log n) time / operation

answer are lists ⇒ not O(1) space ⇒ O(log n + k) running time and k = output size