Data structures are

FUNDAMENTAL!

- All fields of CS involve storing, retrieving and processing data

- Information retrieval
- Geographic Inf. Systems
- Machine Learning
- Text/String processing
- Computer graphics
- ...

Basic elements in study of data structures

- Modeling: How real-world objects are encoded
- Operations: Allowed functions to access + modify structure
- Representation: Mapping to memory
- Algorithms: How are ops. performed?

Course Overview:

- Fundamental data structures + algorithms
- Mathematical techniques for analyzing them
- Implementation

Introduction to Data Structures

- Elements of data structures
- Our approach
- Short review of asymptotics

Our approach:

- Theoretical: Algorithms + Asymptotic Analysis
- Practical: Implementation + practical efficiency

Common:

$O(1)$: constant time [Hash map]

$O(\log n)$: log time (very good!) [Binary search]

$O(n^p)$: (p = constant) Poly time [Geometric search]

e.g. $O(\sqrt{n})$

Asymptotic: "Big-O"

- Ignore constants
- Focus on large n

$$T(n) = 34n^2 + 15n \cdot \log n + 143$$
$$T(n) = O(n^2)$$

Asymptotic Analysis:

- Run time as a function of n ← no. of items
- Worst-case, average-case, randomized
- Amortized: Average over a series of ops.

Basic Data Structures I

- ADTs
- Lists, Stacks, Queues
- Sequential Allocation

Abstract Data Type (ADT)

- Abstracts the functional elements of a data structure (math) from its implementation (algorithm / programming)

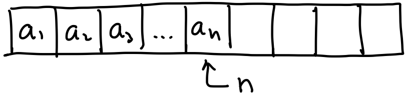
Linear List ADT:

Stores a sequence of elements $\langle a_1, a_2, \dots, a_n \rangle$. Operations:

- init() - create an empty list
- get(i) - returns a_i
- set(i,x) - sets ith element to x
- insert(i,x) - inserts x prior to ith (moving others back)
- delete(i) - deletes ith item (moving others up)
- length() - returns num. of items

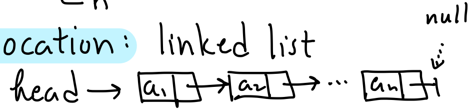
Implementations:

Sequential: Store items in an array

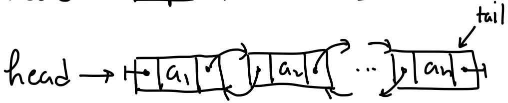


Linked allocation: linked list

Singly:

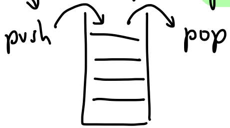


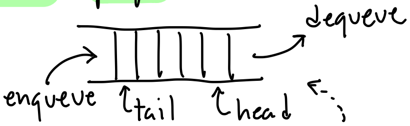
Doubly:



Performance varies with implementation

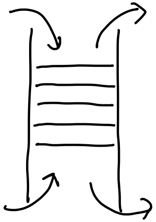
Stack: All access from one side (top) - push + pop





Queue: FIFO list: enqueue inserts at tail and dequeue deletes from head

Deque ("deck"): Can insert or delete from either end



Dynamic Lists + Sequential Allocation: What to do when your array runs out of space?

Doubling Reallocation:

When array of size n overflows

- allocate new array size 2n
- copy old to new
- remove old array

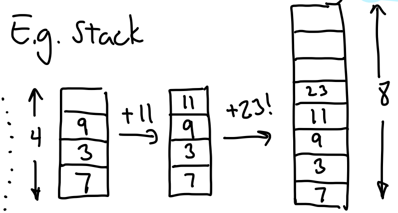
Basic Data Structures II

- Amortized analysis of dynamic stack

Dynamic (Sequential) Allocation

- When we overflow, double

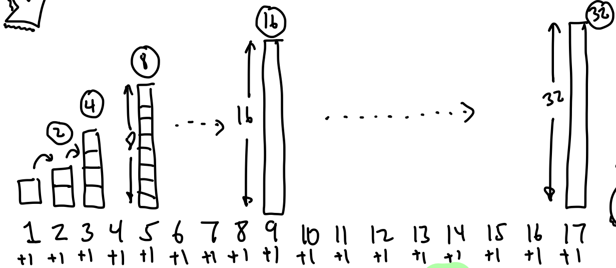
E.g. Stack



Cost model (Actual cost)

Cheap: No reallocation → 1 unit

Expensive: Array of size n is reallocated to size 2n } → 2n+1



Total = 17+(2+4+8+16+32)= 79

Amortized Cost:

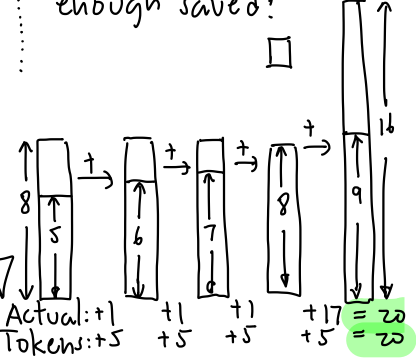
Starting from an empty structure, suppose that any sequence of m ops takes time T(m). The amortized cost is T(m)/m.

Thm: Starting from an empty stack, the amortized cost of our stack operations is at most 5.

[ie. any seq. of m ops has cost ≤ 5·m]

Charging Argument:

- Each request of push/pop we charge user 5 "work tokens"
- We use 1 token to pay for the operation + put other 4 in bank account.
- Will show there is enough in bank account to pay actual costs.



Proof:

- Break the full sequence after each reallocation → run

  1 2 | 3 | 4 5 | 6 7 8 9 | 10 11 ... 16 17 |
- At start of a run there are n+1 items in stack and array size is 2n
- There are at least n ops before the end of run
- During this time we collect at least 5n tokens
  - → 1 for each op
  - → 4 for deposit
- Next reallocation costs 4n, but we have enough saved!

□

Basic Data Structures III

- Dynamic Stack - Wrap-up
- Multilists + Sparse Matrices

Dynamic Stack:

- Showed doubling ⇒ Amortized $\mathcal{O}(1)$
- Other strategies?

Fixed Increment: Increase by a fixed constant

$n \longrightarrow n+100$

Fixed factor: Increase by a fixed constant factor (not nec. 2)

$n \rightarrow 5 \cdot n$

Squaring: Square the size (or some other power)

$n \rightarrow n^2$ or $n \rightarrow \lceil n^{1.5} \rceil$

Which of these provide $\mathcal{O}(1)$ amortized cost per operation?

Leave as exercise ☹

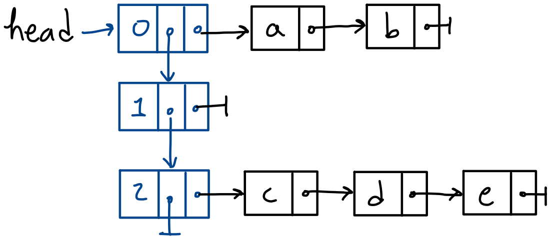
(Spoiler alert!)

Fixed increment → no

Fixed factor → yes

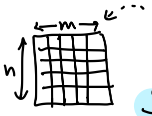
Squaring → ?? (depends on cost model)

Multilists: Lists of lists



Sparse Matrices:

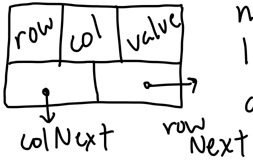
An n×m matrix has n·m entries and takes (naively) $\mathcal{O}(n \cdot m)$ space

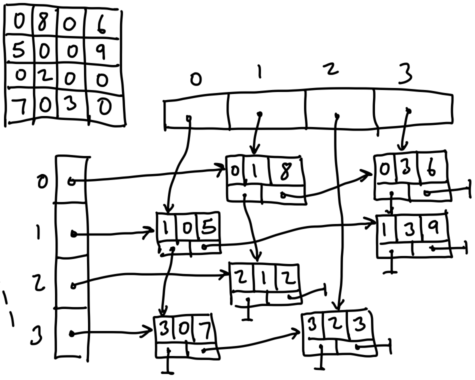


Sparse matrix: Most entries are zero

Idea: Store only nonzero entries linked by row and column

Node:

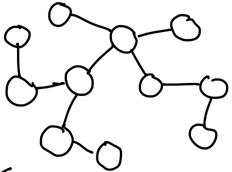




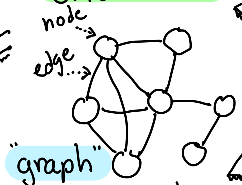
0	8	0	6
5	0	0	9
0	2	0	0
7	0	3	0

Trees: Basic Concepts and Definitions

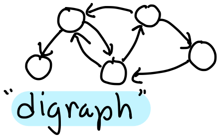
Tree (or "**Free Tree**")
- undirected
- connected
- acyclic graph



Undirected



Directed



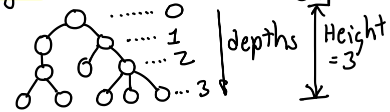
Graph: $G=(V,E)$

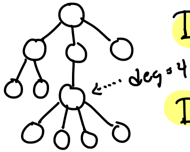
V = finite set of vertices (nodes)

E = set of edges (pairs of vertices)

Depth: path length from root

Height: (of tree) max depth

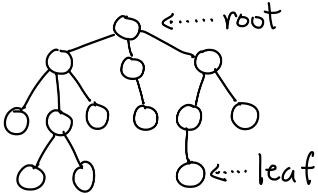




Degree (of node): number of children

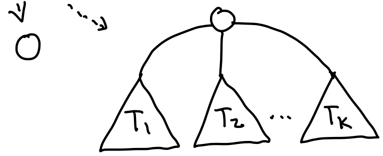
Degree (of tree): max. degree of any node

Rooted tree: A free tree with root node

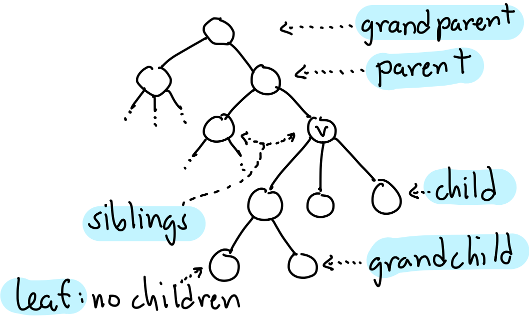


Formal definition:

Rooted tree: is either
- single node (root)
- set of one or more rooted trees ("subtrees") joined to a common root



"Family" Relations

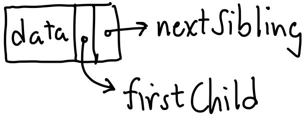


Trees Representation + Binary Trees (1)

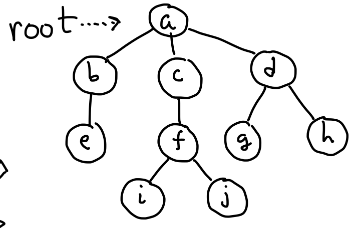
Representing rooted trees:

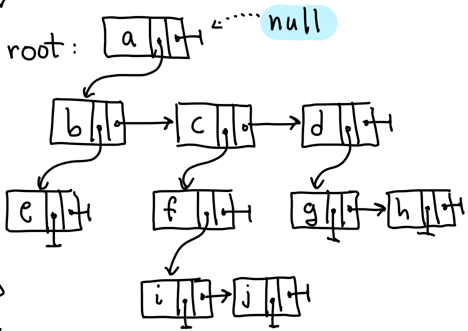
Each node stores a (linked) list of its children

Node structure:



E.g.

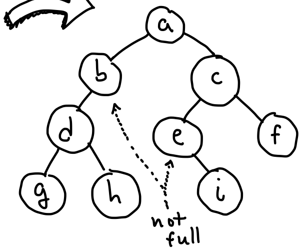




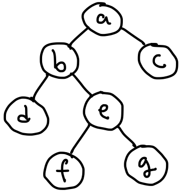
called the **Binary representation**

Binary tree: A rooted tree of degree 2, where each node has two children (possibly null) left + right

(Not full)

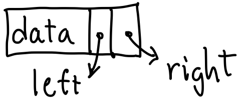


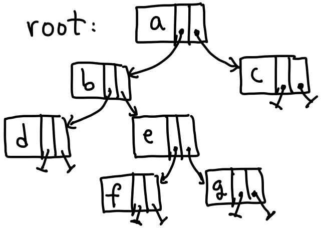
Full:



Full: Every non-leaf node has 2 children

Node structure:





In Java:

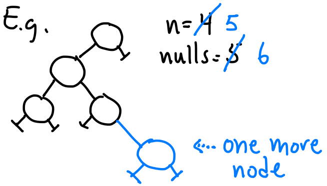
```
class BTNode<E> {
    E data;
    BTNode<E> left;
    BTNode<E> right;
    ....
}
```

generic data

Wasted space?

Theorem: A binary tree with n nodes has n+1 null links

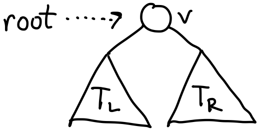
E.g.



Binary Trees: Traversals, Extension, and More

Traversals: How to (systematically) visit the nodes of a rooted tree?

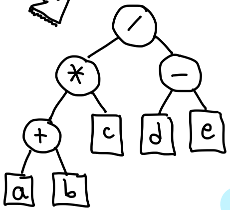
Binary Tree Traversals (can be generalized)



- process/visit v
- traverse T_L
- traverse T_R

recursive

```
traverse(BTNode v) {
    if (v == null) return;
    visit/process v      <- Preorder
    traverse(v.left)
    visit/process v      <- Inorder
    traverse(v.right)
    visit/process v      <- Postorder
}
```

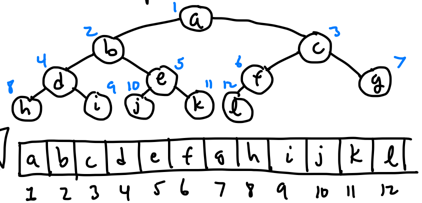


Preorder: / * + a b c - d e

Postorder: a b + c * d e - /

Inorder: a + b * c / d - e

Complete Binary Tree: All levels full (except last)



$\text{parent}(i) = \lfloor i/2 \rfloor$

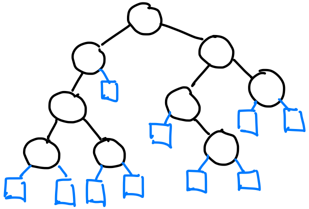
$\text{left}(i) = 2 \cdot i$

$\text{right}(i) = 2 \cdot i + 1$

Challenge: Nonrecursive traversals

Those wasteful null links....

Extended binary tree: Replace each null link with a special leaf node: **external node**



Thm: An extended binary tree with n internal nodes (black) has n+1 external nodes (blue)

Observation: Every extended binary tree is **full**

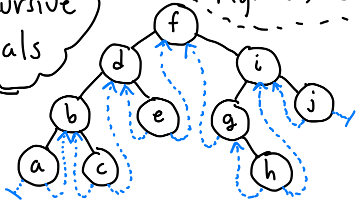
Another way to save space...

Threaded binary tree: Store (useful) links in the null links. (Use a mark bit to distinguish link types.)

E.g. **Inorder Threads:**

Null left → inorder predecessor

Null right → " successor



Binary Search Trees I

- Basic definitions
- Finding keys

Search: Given a set of n **entries** each associated with **key** x_i and **value** v_i

- Store for quick access + updates
- **Ordered**: Assume that keys are totally ordered: <, >, ==

Dictionary:

insert(Key x, Value v)

- insert (x, v) in dict. (No duplicates)

delete(Key x)

- delete x from dict. (Error if x not there)

find(Key x)

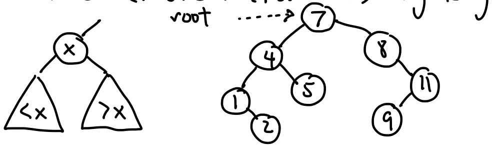
- returns a reference to associated value v, or **null** if not there.

Sequential Allocation?

- Store in array sorted by key
  - → **Find**: O(log n) by binary search
  - → **Insert/Delete**: O(n) time

Can we achieve O(log n) time for all ops? ☆ **Binary Search Trees** ☆

Idea: Store entries in binary tree sorted (inorder traversal) by key

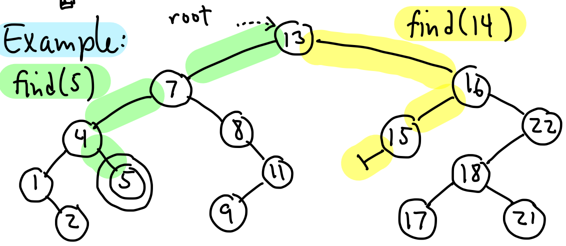


Find: How to find a key in the tree?

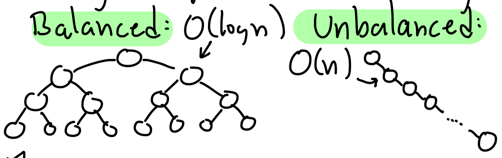
- Start at root p ← root
- if (x < p.key) search left
- if (x > p.key) search right
- if (x == p.key) found it!
- if (p == null) not there!

```
Value find(Key x, BSTNode p){
    if (p == null) return null
    else if (x < p.key)
        return find(x, p.left)
    else if (x > p.key)
        return find(x, p.right)
    else return p.value
}
```

Example:

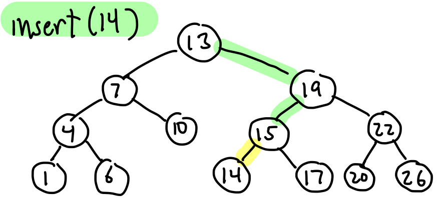


Efficiency: Depends on tree's height



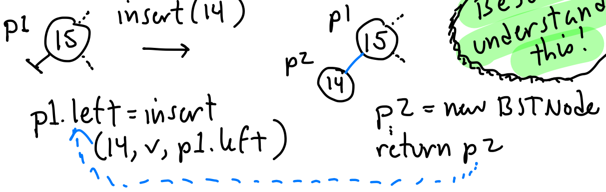
Binary Search Trees II
- insertion
- deletion

Insert (Key x, Value v)
- find x in tree
- if found ⇒ error! duplicate key
- else: create new node where we "fell out"



```
BSTNode insert(Key x, Value v, BSTNode p){
    if (p==null)
        p = new BSTNode(x,v)
    else if (x < p.key)
        p.left = insert(x,v, p.left)
    else if (x > p.key)
        p.right = insert(x,v, p.right)
    else throw exception → Duplicate!
    return p
}
```

Why did we do: p.left = insert(x,v, p.left)?



p1.left = insert (14, v, p1.left)

p2 = new BSTNode
return p2

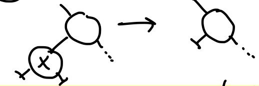
Be sure you understand this!

Delete (Key x)
- find x
- if not found → error
- else: remove this node + restore BST structure

How?

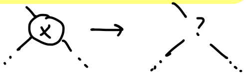
3 cases:

① ⓧ is a leaf



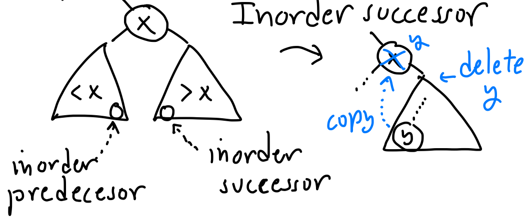
② ⓧ has single child

3. ⓧ has two children



Find replacement node ⓨ, copy to ⓧ, and then delete ⓨ

Replacement Node?



Binary Search Trees III

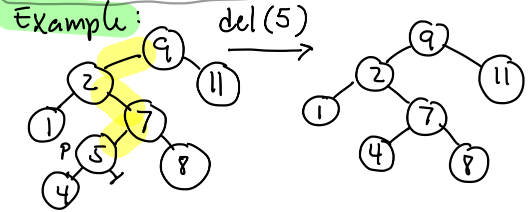
- deletion
- analysis
- Java

```
BSTNode delete(Key x, BSTNode p){
    if(p == null) ERROR! Key not found
    else
        if(x < p.key)
            p.left = delete(x, p.left)
        else if (x > p.key)
            p.right = delete(x, p.right)
        else if (either p.left or p.right null)
            if (p.left == null)
                return p.right
            if (p.right == null)
                return p.left
        else
            r = findReplacement(p)
            copy r's contents to p
            p.right = delete (r.key, p.right)
    return p
}
```

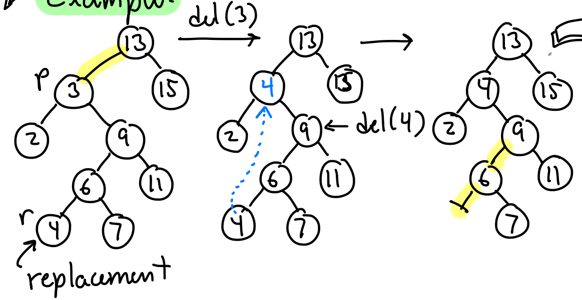
Find Replacement Node

```
BSTNode findReplacement(BSTNode p){
    BSTNode r = p.right
    while(r.left ≠ null)
        r = r.left
    return r
}
```

Example:



Example:



Analysis:

All operations (find, insert, delete) run in O(h) time, where h = tree's height

But height can vary from O(log n) to O(n)...

Expected case is good

Thm: If n keys are inserted in random order, expected height is O(log n).

Java Implementation:

- Parameterize Key + Value types: extends Comparable
  class BinSearchTree<K,V>{...
- BSTNode - inner class
- Private data: BSTNode root
- insert, delete, find: local
- provide public fns
  insert, delete, find

Java implementation (see notes for details)

```
public class BSTree<Key extends Comparable, Value> {
    class Node {
        Key key
        Value value
        Node left, right

        .... constructor, toString...
    }
```

Inner class for node (protected)

```
    Value find(Key x, Node p) {...}
    Node insert(Key x, Value v, Node p) {...}
    Node delete(Key x, Node p) {...}
```

Local helpers (private or protected)

```
    private Node root;
```

Data (private)

```
    public Value find(Key x) {...}
    public void insert(Key x, Value v) {...}
    public void delete(Key x) {...}
}
```

Public members (invoke helpers)

AVL Trees I

- Basic defs
- Height props
- Rotations

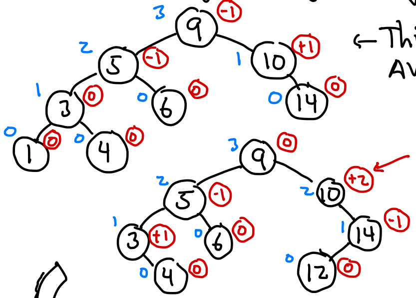
AVL Height Balance

- for each node v, the heights of its subtrees differ by ≤ 1.

AVL tree: A binary search tree that satisfies this condition

Balance factor:

$$\mathrm{bal}(v) = \mathrm{hgt}(v.\mathrm{right}) - \mathrm{hgt}(v.\mathrm{left})$$



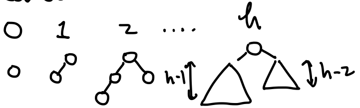
← This is an AVL tree

Not an AVL tree

Does this imply O(log n) height?

Worst cases:

height: $h =$ 0 1 2 .... h



nodes: $n =$ 1 2 4 7 12 20 ...

$n+1 =$ 2 3 5 8 13 21 ...

Recall: $F_0 = 0$, $F_1 = 1$, $F_h = F_{h-1} + F_{h-2}$

Conjecture: Min no. of nodes in AVL tree of height h is $F_{h+3} - 1$

Theorem: An AVL tree of height h has at least $F_{h+3} - 1$ nodes.

Proof: (Induct. on h)

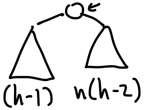
$h = 0$: $n(h) = 1 = F_3 - 1$

$h = 1$: $n(h) = 2 = F_4 - 1$

$h \geq 2$: $n(h) = 1 + n(h-1) + n(h-2)$

$= 1 + (F_{h+2} - 1) + (F_{h+1} - 1)$

$= (F_{h+2} + F_{h+1}) - 1 = F_{h+3} - 1$ □



Corollary: An AVL tree with n nodes has height O(log n)

Proof: Fact: $F_h \approx \varphi^h / \sqrt{5}$ where

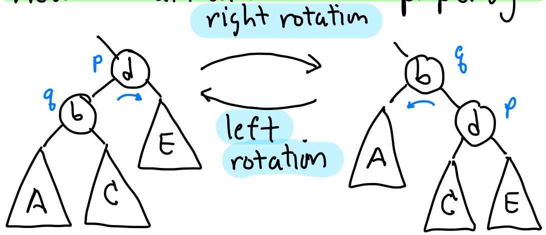
$\varphi = (1+\sqrt{5})/2$ "Golden ratio"

$n \geq \varphi^{h+3} = c \cdot \varphi^h \Rightarrow h \leq \log_\varphi n + c'$

$\Rightarrow h \leq \log_2 n / \log_2 \varphi$

$= O(\log n)$ □

How to maintain the AVL property?



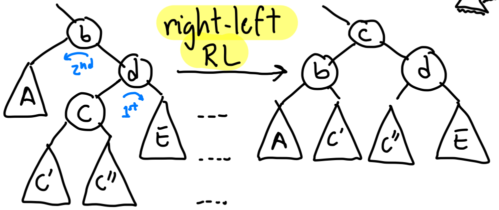
$A < b < C < d < E$ $A < b < C < d < E$

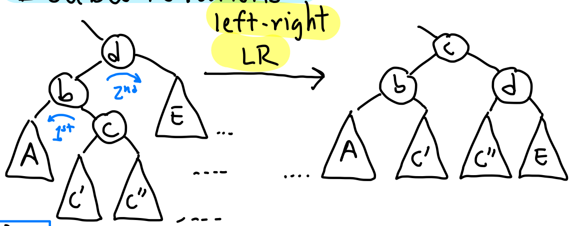
```
BSTNode rotateRight(BSTNode p){
    BSTNode q = p.left
    p.left = q.right
    q.right = p
    return q
}
```

AVL Trees II

- double rotations
- insertion

Double rotations:





```
BSTNode rotateLeftRight(BSTNode p)
    p.left = rotateLeft(p.left)
    return rotateRight(p)
```

AVL Tree:

AVL Node: Same as BSTNode (from Lect 4) but add: **int height**

simpler than bal factor

Utilities:

```
int height(AVLNode p)
    return { p == null → -1
           { o.w.      → p.height

void updateheight(AVLNode p)
    p.height = 1 + max(height(p.left),
                       height(p.right))

int balanceFactor(AVLNode p)
    return height(p.right)-
           height(p.left)
```

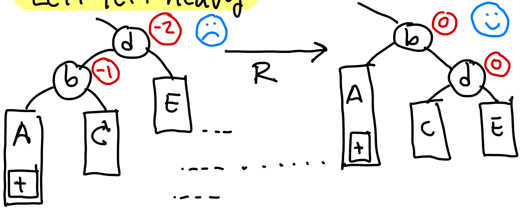
Find: Same as BST.

Insert: Same as BST but as we "back out" rebalance

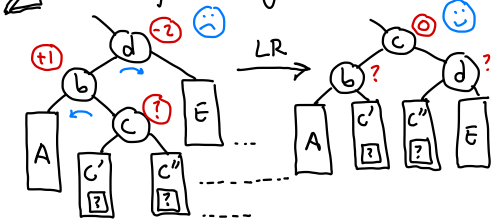
How to rebalance?

Bal=-2

Left-left heavy



Left-right heavy:



```
AVLNode insert(Key x, Value v, AVLNode p){
    if (p == null) p = new AVLNode(x,v)
    else if (x < p.key)
        p.left = insert(x,v,p.left)
    else if (x > p.key)
        p.right = insert(x,v,p.right)
    else throw -Error-Duplicate!
    return rebalance(p)
```

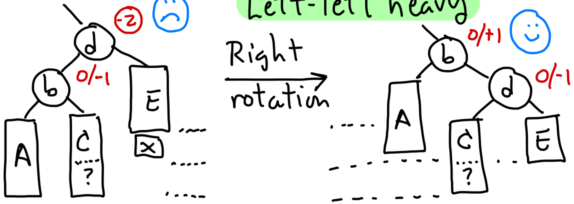
```
AVLNode rebalance(AVLNode p)
    if (p==null) return p
    if (balanceFactor(p) < -1)
        if (ht(p.left.left) ≥ ht(p.left.right))
            p = rotateRight(p)
        else p = rotateLeftRight(p)
    else if (balanceFact(p) > +1)
        ...(symmetrical)
    updateHeight(p); return p
```

AVL Trees III

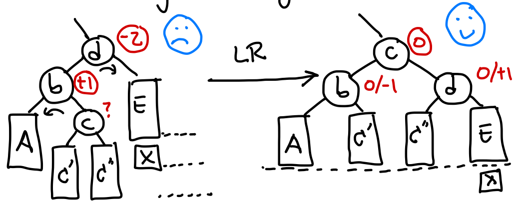
- Deletion
- Examples

Cases: Balance factor -2

Left-left heavy

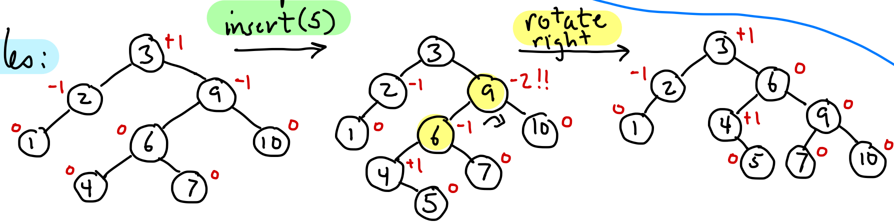


Left-right heavy



```
AVLNode delete (Key x, AVLNode p)
    :
    same as BST delete
    :
    return rebalance(p)
```

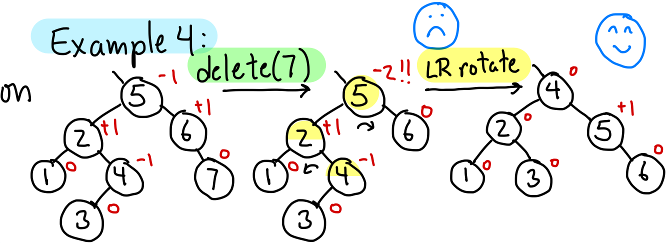
Examples:



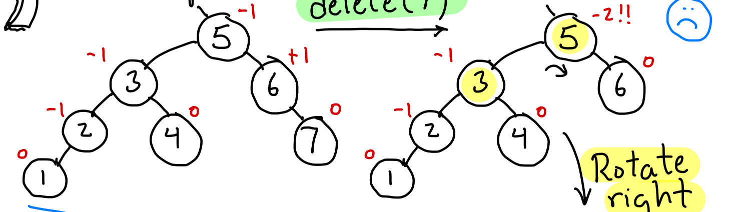
Deletion: Basic plan

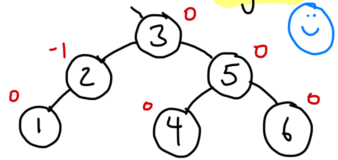
- Apply standard BST deletion
  - find key to delete
  - find replacement node
  - copy contents
  - delete replacement
  - rebalance

Example 4:

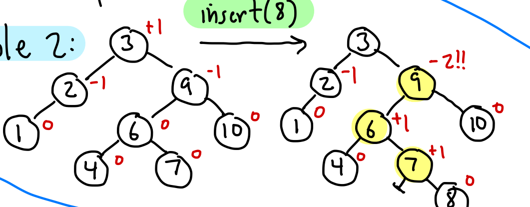


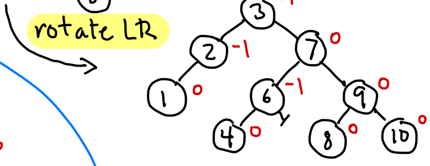
Example 3:





Example 2:



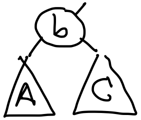


2-3 Trees I

Node types:

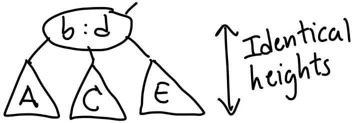
2-Node

1 key
2 children



3-Node

2 keys
3 children

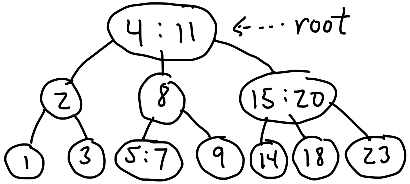


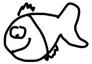
Def: A **2-3 tree** of height h is either:

- Empty ($h=-1$)
- A 2-Node root and two subtrees, each 2-3 tree of height $h-1$
- A 3-Node root and three subtrees... height $h-1$.

Example:

2-3 tree of height 2

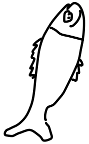




Recap:

AVL: Height balanced Binary

2-3 tree: Height exact Variable width

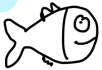


Thm: A 2-3 tree of n nodes has height $O(\log n)$

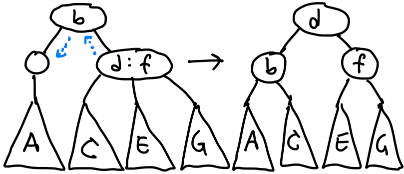
Roughly: $\log_3 n \le h \le \log_2 n$

How to maintain balance?

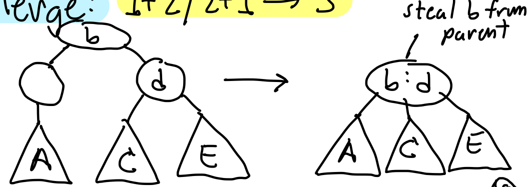
- Split
- Merge
- Adoption (Key rotation)



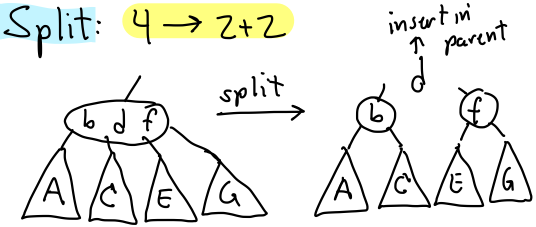
Adoption (Key-Rotation) $1+3 = 2+2$



Merge: $1+2/2+1 \to 3$

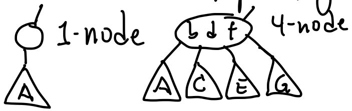


Split: $4 \to 2+2$



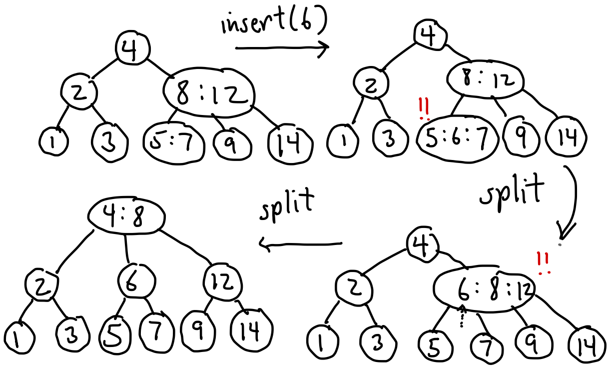
Conceptual tool:

We'll allow 1-nodes + 4-nodes temporary

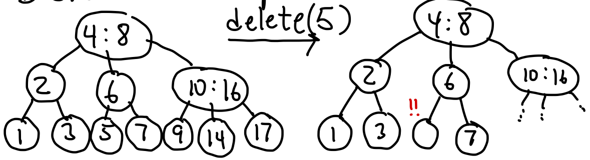


2-3 Trees II

Insertion example:

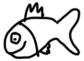


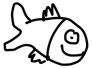
Delete Example:



Deletion remedy:

- Have a 3-node neighboring sibling → adopt
- o.w.: Merge with either siblings + steal key from parent





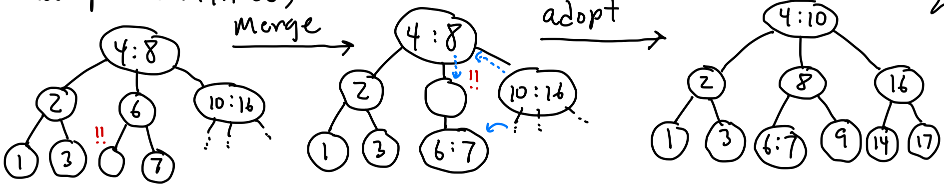
Dictionary operations:

Find - straightforward

Insert - find leaf node where key "belongs" + add it (may split)

Delete - find/replacement/ merge or adopt

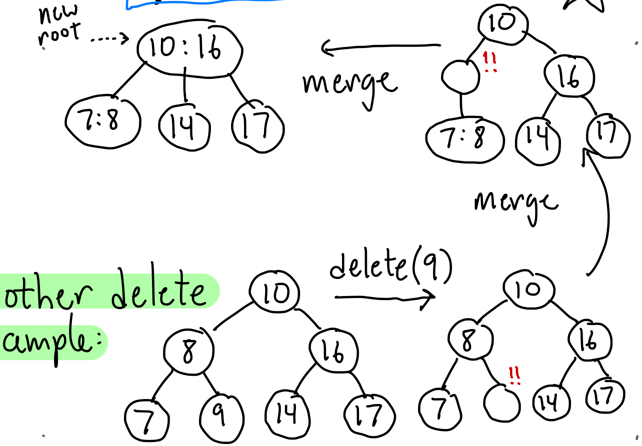
Example (continued)



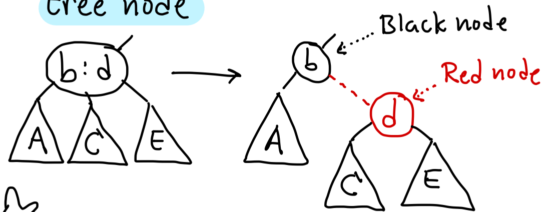
Implementation?

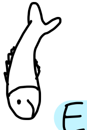
```
class TwoThreeNode {
    int nChildren
    TwoThreeNode children[3]
    Key key[2]
}
```

Another delete example:

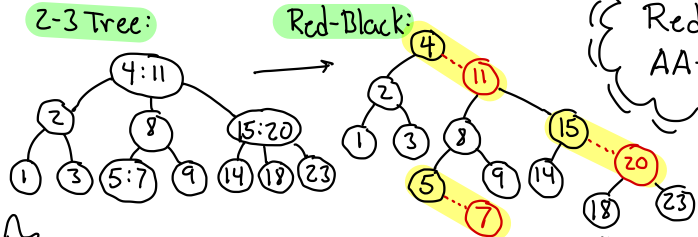


Encoding 3-node as binary tree node





Example:



Rules:

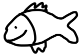
① Every node labeled red/black

② Root is black

③ Nulls treated as if black

④ If node is red, both children are black

⑤ Every path from root to null has same no. of black



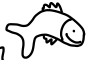
Some history:

2-3 Trees: Bayer 1972

Red-black Trees: Guibas & Sedgewick 1978 (a binary variant of 2-3)

Rumor - Guibas had two pens - red & black to draw with

Red-Black and AA-Trees I



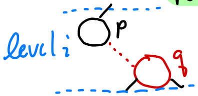
Lemma: A red-black tree with n keys has height $O(\log n)$

Proof: It's at most twice that of a 2-3 tree.

Q: Is every Red-Black Tree the encoding of some 2-3 tree?

AA-Trees: Simpler to code

- **No null pointers**: Create a **sentinel node**, nil, and all nulls point to it → **nil**:



- **No colors**: Each node stores **level number**. Red child is at same level as parent.

q is red ⟺ q.level == p.level

What we need are stricter rules!

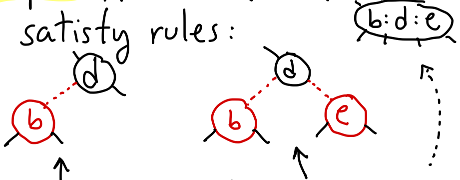
AA-tree:

Arne Anderson 1993

New rule:

⑥ Each red node can arise only as right child (of a black node)

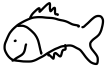
Nope! Alternatives that satisfy rules:



A "**left-skewed**" encoding

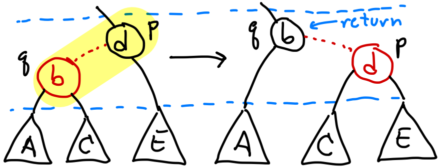
Corresponds to **2-3-4 trees**

Red-Black + AA Trees II



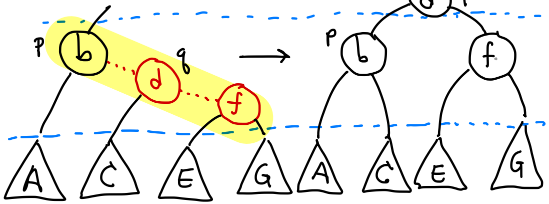
Restructuring Ops:

Skew: Restore right skew
→ If black node has red left child, rotate



How to test? p.left.level == p.level

Split: If a black node has a right-right red chain, do a left rotation at p (bringing its right child q up) and move q up one level.

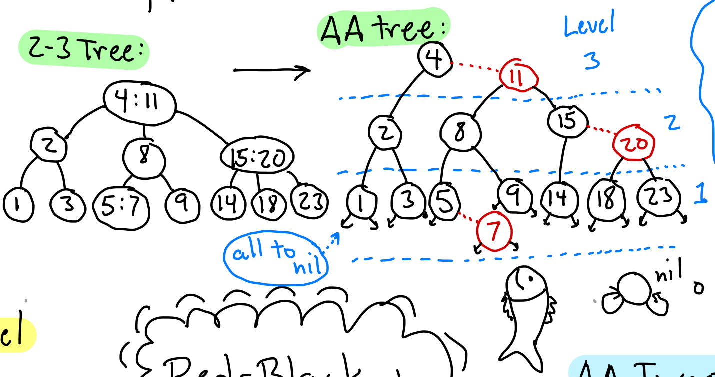


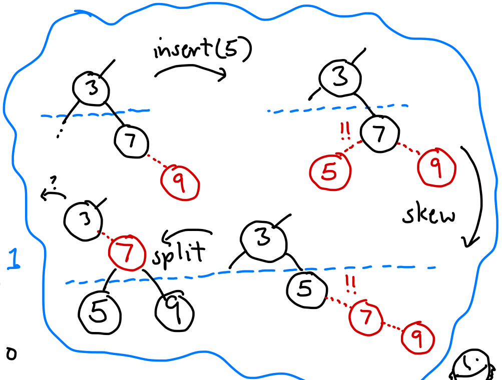
what 2-3 op does this remind you of ?

How to test? p.level == p.right.level == p.right.right.level

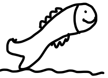
not needed (levels are monotone)

Example:



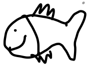


AA Insertion:
- Find the leaf (as usual)
- Create new red node
- Back out applying skew + split



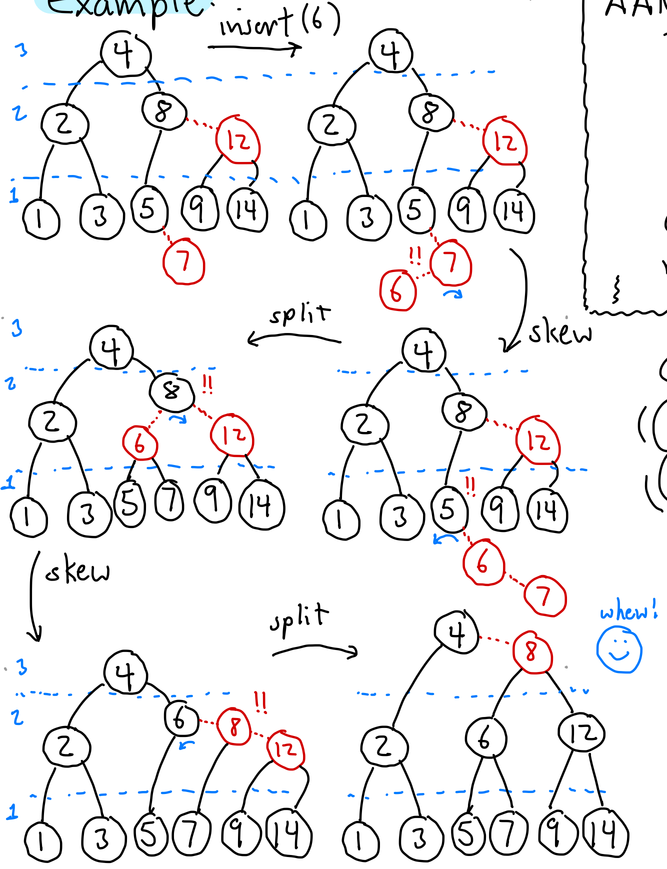
```
AANode skew(AANode p) {
    if (p == nil) return p
    if (p.left.level == p.level) {        right rotate p
        AANode q = p.left                 ←
        p.left = q.right ; q.right = p
        return q  ← new subtree root
    } else return p  ← everything's fine
}
```

```
AANode split(AANode p) {
    if (p == nil) return p
    if (p.right.right.level == p.level) {
        AANode q = p.right
        p.right = q.left      ← left rotation at p
        q.left = p
        q.level += 1  ← move q up a level
        return q
    } else return p  ← all okay
}
```

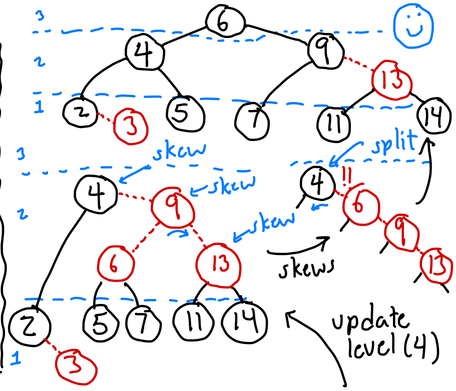


Red-Black and AA Trees III

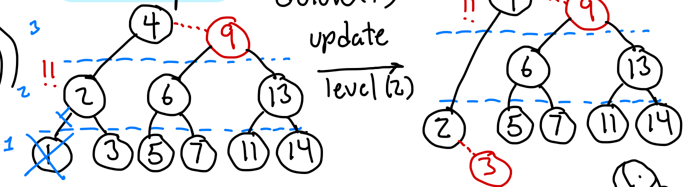
Example:

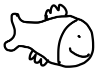


```
AANode insert(Key x, Value v, AANode p) {
                                  leaf level   left, right
    if (p == nil)
        p = new AANode(x, v, 1, nil, nil)
    else if (x < p.key) ... insert on left
    else if (x > p.key) ... insert on right
    else Duplicate Key!
    return split(skew(p))
}
```



Example: delete(1)

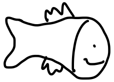




Deletion:

Two more helpers:

updateLevel: If p's level exceeds $\ell = 1 + \min(p.left.level, p.right.level)$ then set p's level to ℓ & also p's right child



fixAfterDelete(p):

- update p's level
- skew(p), skew(p.right)
  skew(p.right.right)
- split(p), split(p.right)

deletion: Same as AVL deletion, but end with:
return fixAfterDelete(p)

Randomized Data Structures

Treaps I

History:

1989: Seidel + Aragon
[Explosion of randomized algorithms]

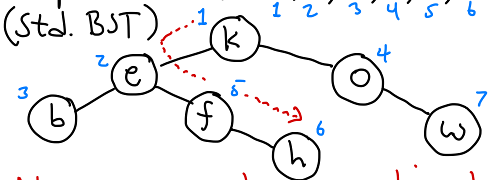
Later discovered this was already known: Priority Search Trees from different context (geometry)
McCreight 1980

Intuition:

- Random insertion into BSTs
  $\Rightarrow O(\log n)$ expected height
- Worst case can be very bad
  $O(n)$ height
- Treap: A tree that behaves as if keys are inserted in random order

Example: Insert: k, e, b, o, f, h, w
1 2 3 4 5 6 7

(Std. BST)



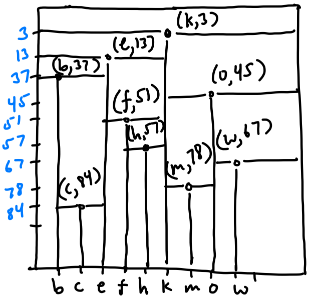
Along any path - Insertion times increase

- Use a random number generator
- Running in expectation over all random choices
- Often simpler than deterministic

Obs: In a standard BST, keys are by inorder + insert times are in heap order (parent < child)

Geometric Interpretation:

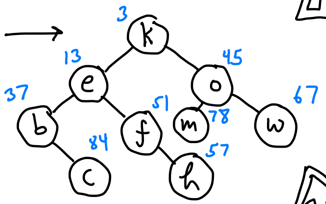
key → x
priority → y



McCreight's Priority Search Tree

Example:

Key	Priority
b	37
c	84
e	13
f	51
h	57
k	3
m	78
o	45
w	67



Treap: Each node stores a key + a random priority.
Keys are in inorder.
Priorities are in heap order

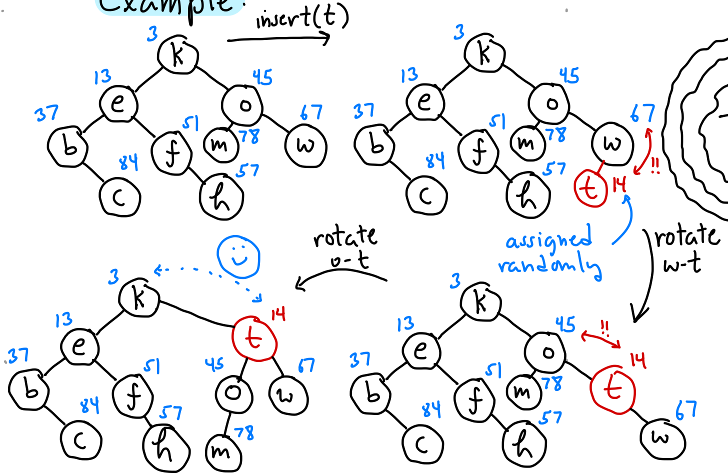
? Is it always possible to do both?

Yes: Just consider the corresponding BST

Treaps II

Insertion: As usual, find the leaf + create a new leaf node.
- Assign random priority
- On backing out - check heap order + rotate to fix.

Example:

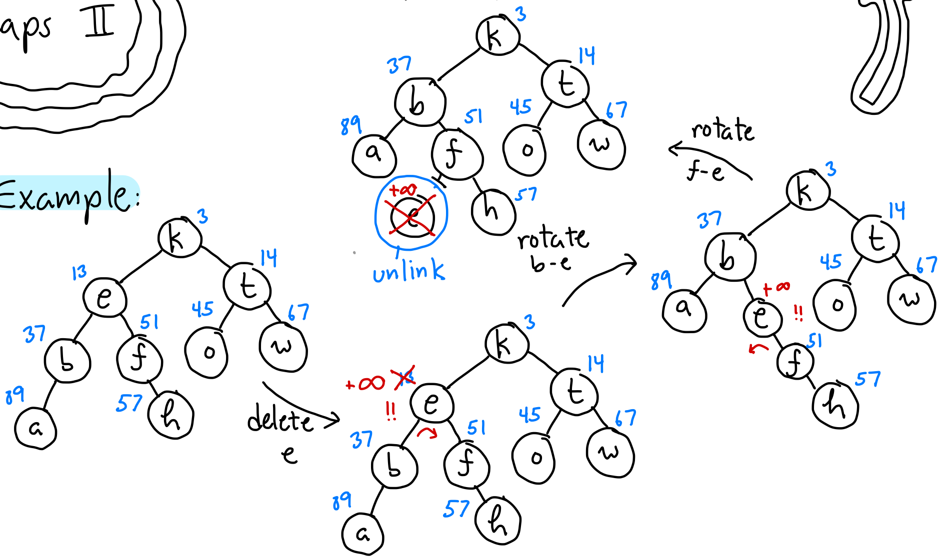


Deletion: (Cute solution) Find node to delete. Set its priority to +∞. Rotate it down to leaf level + unlink.

Theorem: A treap containing n entries has height O(log n) in expectation (averaged over all assignments of random priorities)

Proof: Follows directly from BST analysis

Example:

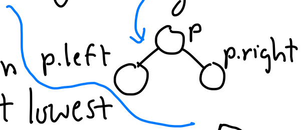


Implementation: (See pdf notes)

Node: Stores priority + usual...

Helpers:

lowest priority(p)
returns node of lowest priority among:



restructure:
performs rotation (if needed) to put lowest priority node at p.

Skip Lists I

Ideal Skip List:

- Organize list in levels
- Level 0: Everything
  - 1: Every other
  - 2: Every fourth
  - ⋮
  - i: Every 2^i

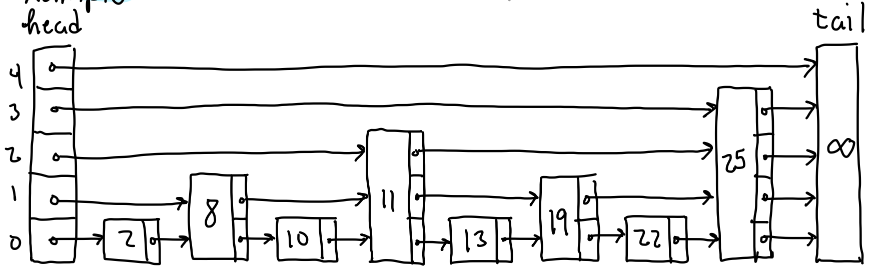
Sorted linked lists:

- Easy to code
- Easy to insert/delete
- Slow to search… $O(n)$

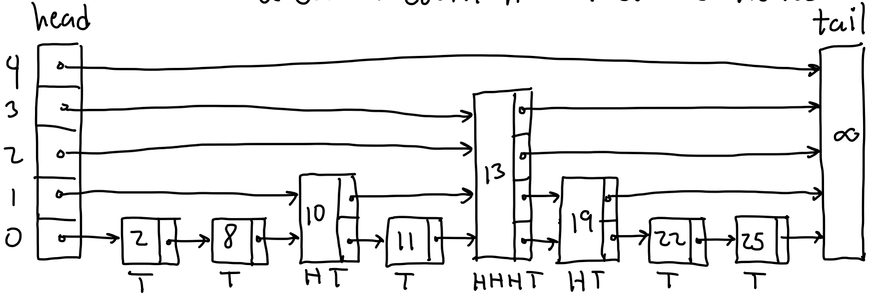
Idea: Add extra links to skip

How to generalize?

Example:



Too rigid → **Randomize!** To determine level – toss a coin + count no. of consec. heads:



Node Structure: (Variable sized)

```
class SkipNode {
    Key key
    Value value
    SkipNode[] next
}
```

In constructor, set size (height)

```
Value find(Key x) {
    i = topmost Level
    SkipNode p = head
    while (i ≥ 0) {
        if (p.next[i].key ≤ x) p=p.next[i]
        else i--
    }
    if (p.key == x) return p.value
        else return null
}
```

current node

until we hit base level

advance horizontal

drop down a level

we are at base level

Skip Lists II

Thm: A skip list with n nodes has O(log n) levels in expectation

Proof: Will show that probability of exceeding $c \cdot \lg n$ is $\leq 1/n^{c-1}$.

→ Prob that any given node's level exceeds ℓ is $1/2^{\ell}$ [ℓ consecutive heads]

→ Prob that any of n node's level exceeds ℓ is $\leq n/2^{\ell}$ [n trials with prob $1/2^{\ell}$]

→ Let $\ell = c \cdot \lg n$ ($\lg \equiv \log_2$)

Prob that max level exceeds $c \cdot \lg n$ is:

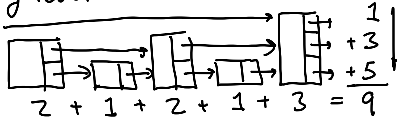
$$\begin{aligned} &\leq n/2^{\ell} = n/2^{(c \cdot \lg n)} \\ &= n/(2^{\lg n})^{c} \\ &= n/n^{c} = 1/n^{c-1} \end{aligned}$$

☐

Obs: Prob. level exceeds $3 \cdot \lg n$ is $\leq 1/n^2$. (If $n \geq 1{,}000$, chances are less than 1 in million!)

Thm: Total space for n-node skip list is O(n) expected.

Proof: Rather than count node by node, we count level by level:



- Let n_i = no. of nodes that contrib. to level i.
- Prob that node at level $\geq i$ is $1/2^i$
- Expected no. of nodes that contrib. to level $i = n/2^i$

$\Rightarrow E(n_i) = n/2^i$

Total space (expected) is:

$$E\left(\sum_{i=0}^{\infty} n_i\right) = \sum_{i=0}^{\infty} E(n_i) = \sum_{i=0}^{\infty} n/2^i$$

$$= n \sum_{i=0}^{\infty} 1/2^i = 2n$$

☐

Thm: Expected search time is O(log n)

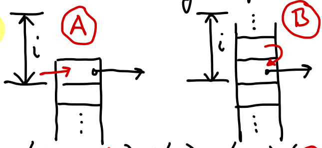
Proof:

- We have seen no. levels is O(log n)
- We'll show that we visit 2 nodes per level on average

Obs – Whenever search arrives first time to a node, its at top level. (Can you see why?)

Def: E(i) = Expect. num. nodes visited among top i levels.

Cases:



$$E(i) = \underset{\text{current node}}{1} + (\text{Prob}(\textcircled{A}))\underset{\text{same level}}{E(i)} + (\text{Prob}(\textcircled{B}))\underset{\text{from prior level}}{E(i-1)}$$

$$= 1 + \tfrac{1}{2}E(i) + \tfrac{1}{2}E(i-1)$$

$$\Rightarrow E(i)(1 - \tfrac{1}{2}) = 1 + \tfrac{1}{2}E(i-1)$$

$$\Rightarrow E(i) = [1 + \tfrac{1}{2}E(i-1)]2 = 2 + E(i-1)$$

Basis: $E(0) = 0 \Rightarrow E(i) = 2 \cdot i$

Let ℓ = max level. Total visited $= E(\ell) = 2 \cdot \ell$

$\Rightarrow$ We visit 2 nodes per level on average.

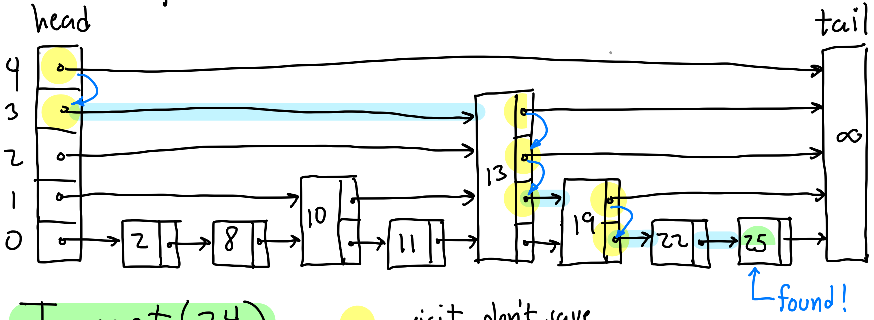
☐

Skip Lists III

Delete:
- Start at top
- Search each level saving last node < key
- On reaching node at level 0, remove it and unlink from saved pointers

Insert: (Similar to linked lists)
- Start at top level
- At each level:
  - Advance to last node ≤ key
  - Save node + drop level
- At level 0:
  - Create new node (flip coins to determine height)
  - Link into each saved node

Example: find(25)

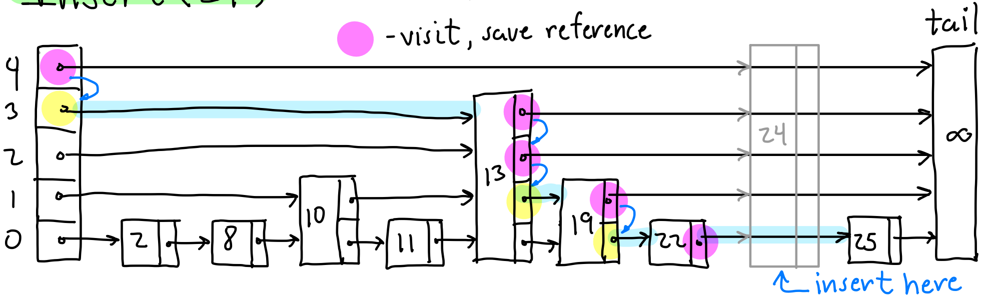


↑ found!

Insert(24)

🟡 - visit, don't save

🟣 - visit, save reference

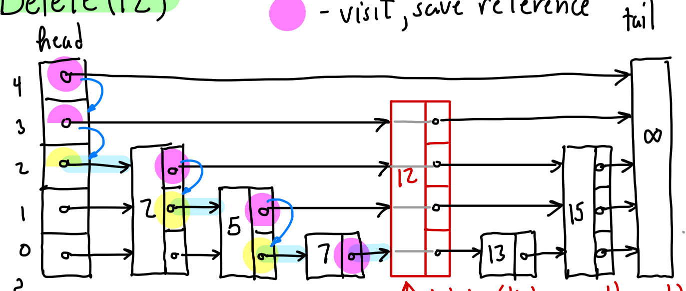


↑ insert here

Delete(12)

🟡 - visit, don't save

🟣 - visit, save reference



↑ delete (links pass through)

Analysis: All operations run in time ~ find ⇒ $O(\log n)$ expected

Note: Variation in running times due to randomness only - not sequence
⇒ User cannot force poor performance.

SPLAY TREES I

Other/Better Criteria?

Expected case: Some keys more popular than others

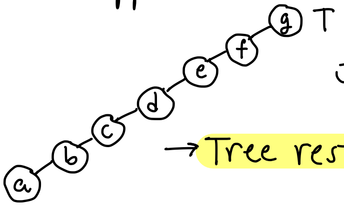
Self-adjusting: Tree adapts as popularity changes

How to design/analyze?

Splay Tree: A self-adjusting binary search tree

- **No rules!** (yay anarchy!)
  - No balance factors
  - No limits on tree height
  - No colors/levels/priorities
- **Amortized efficiency:**
  - Any single op - slow
  - Long series - efficient on avg.

Intuition: Let T be an unbalanced BST + suppose we access its deepest key



find("a")

ugh!

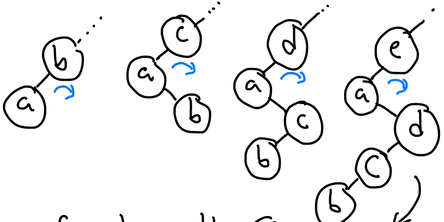
→ Tree restructures itself

Recap: Lots of search trees

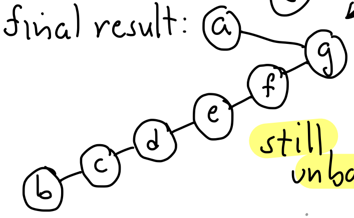
- Unbalanced BSTs
- AVL Trees
- 2-3, Red-black, AA Trees
- Treaps + Skip lists

→ **Focus:** Worst-case or randomized expected case

Idea I: Rotate "a" to top (Future accesses to "a" fast)



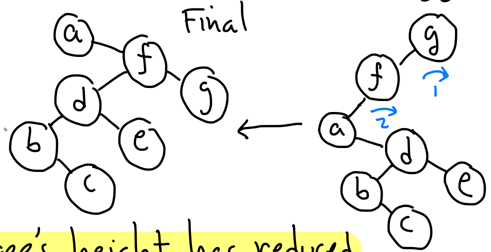
.... final result:



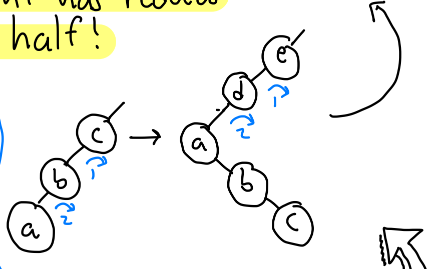
still unbalanced!

Lesson: Different combinations of rotations can:

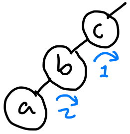
- bring given node to root
- significantly change (improve) tree structure.



Tree's height has reduced by ~ half!



Idea II: Rotate 2 at a time - upper + lower

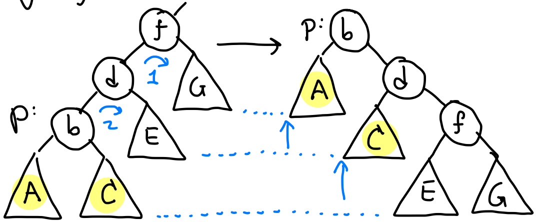


Splay Trees II

Splay(Key x):

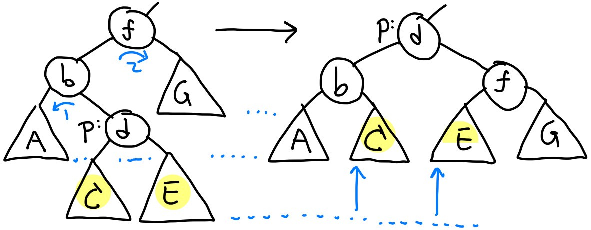
```
Node p ← find x by standard BST search
while (p ≠ root) {
    if (p == child of root) zig(p)
    else /* p has grand parent */
        if (p is LL or RR grand child) zigzig(p)
        else /* p is LR or RL gr. child */ zigzag(p)
```

ZigZig(p): [LL case]



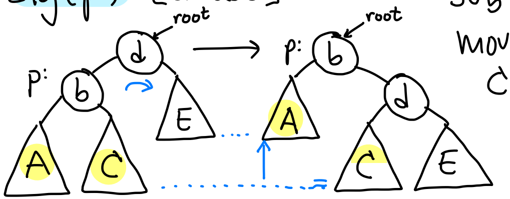
Subtrees A,C move up↑

ZIG ZAG(p): [LR case]



Subtrees C,E of p move up↑

Zig(p): [L case]



Subtree A moves up↑
C unchanged

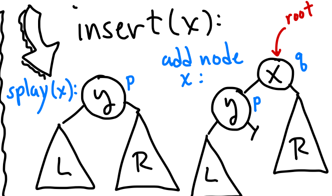
find(x):

```
root ← splay(x)
if (root.key == x)
    return root.value
else return null
```

insert(x):

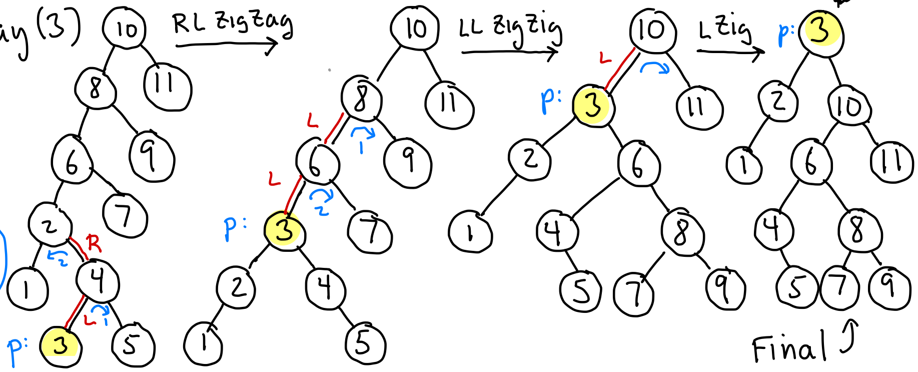
```
Node p ← splay(x)
if (p.key == x)   Error !!
q ← new Node(x)
if (p.key < x)
    q.left ← p
    q.right ← p.right
    p.right ← null
else ... (symmetrical)...
root ← q
```

insert(x):

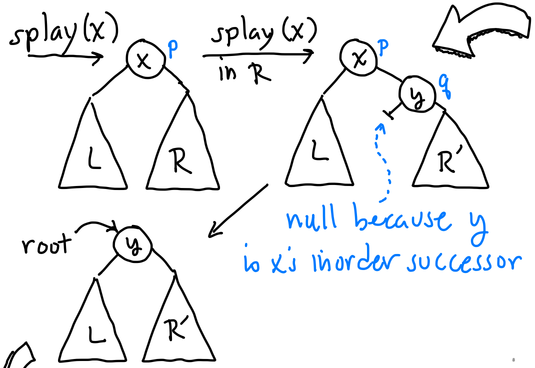


Example:

splay(3)



SPLAY TREES III



delete(x):

```
splay(x) [x now at root]
p = root
if (p.key ≠ x) error!
splay(x) in p's right subtree
q = p.right [q's key is x's successor]
q.left = p.left        [q.left == null]
root = q
```

Dynamic Finger Theorem:

Keys: $x_1 < \dots < x_n$. We perform accesses $x_{i_1}, x_{i_2}, \dots, x_{i_m}$.

Let $\Delta_j = i_j - i_{j-1}$: distance between consecutive items

Thm: Total access time is $O(m + n\log n + \sum_{j=1}^{m}(1 + \lg \Delta_j))$

Analysis:

- Amortized analysis
- Any one op might take $O(n)$
- Over a long sequence, average time is $O(\log n)$ each
- Amortized analysis is based on a sophisticated potential argument
- Potential: A function of the tree's structure
  - Balanced ⇒ Low potential
  - Unbalanced ⇒ High potential
- Every operation tends to reduce the potential

Splay Trees are Amazingly Adaptive!

Balance Theorem: Starting with an empty dictionary, any sequence of m accesses takes total time

$$O(m \log n + n \log n)$$

where n = max. entries at any time.

Static Optimality:

- Suppose key x_i is accessed with prob p_i $(\sum_{i=1}^{n} p_i = 1)$
- Information Theory: Best possible binary search tree answers queries in expected time $O(H)$ where $H = \sum_i p_i \lg 1/p_i$ ← Entropy

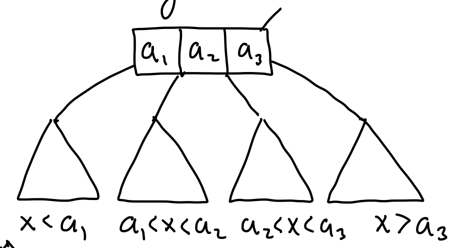
Static Optimality Theorem: Given a seq. of m ops. on splay tree with keys $x_1 \dots x_n$, where x_i is accessed q_i times. Let $p_i = q_i/m$. Then total time is $O(m \sum_i p_i \lg 1/p_i)$

B-Trees I

Secondary Memory:

- Most large data structures reside on disk storage
- Organized in blocks - pages
- Latency: High start-up time
- Want to minimize no. of blocks accessed

Multiway Search Trees:



B-Tree:

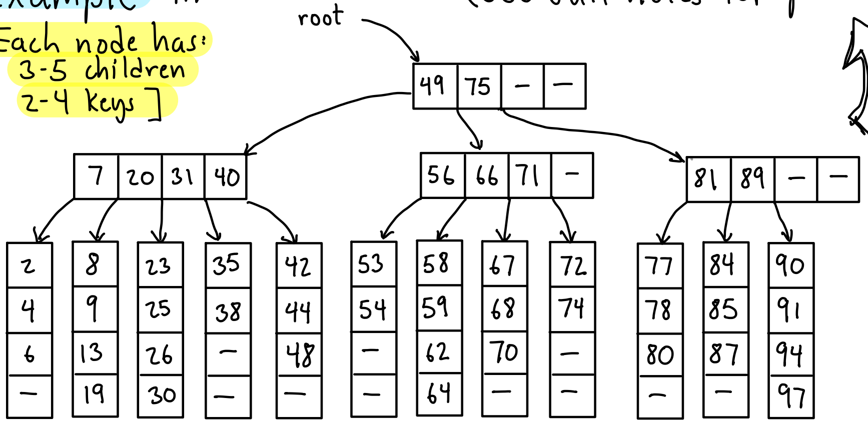
- Perhaps the most widely used search tree
- 1970 - Bayer + McCreight
- Databases
- Numerous variants

B-Tree: of order m (≥3)

- Root is leaf or has ≥ 2 children
- Non-root nodes have $\lceil m/2 \rceil$ to m children [null for leaves]
- k children ⇒ k-1 key-values
- All leaves at same level

Example: m=5

[Each node has:
3-5 children
2-4 keys]



Theorem:

A B-tree of order m with n keys has height at most $(\lg n)/\gamma$, where $\gamma = \lg(m/2)$

(See full notes for proof)

Node Structure:

constant int M=...

```
class BTreeNode {
    int       nChild     // no. of children
    BTreeNode child[M]   // children
    Key   key[M-1]  // keys
    Value value[M-1] // values
}
```

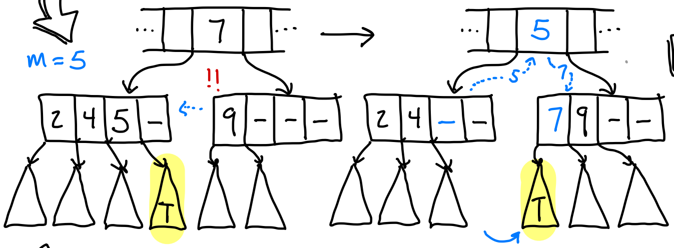
B-Trees II

B-Tree restructuring:

- Generalizes 2-3 restructure
- Key rotation (Adoption)
- Splitting (insertion)
- Merging (deletion)

Key Rotation (Adoption)

- A node has too few children $\lceil m/2 \rceil - 1$
- Does either immediate sibling have extra? $\geq \lceil m/2 \rceil + 1$
- Adopt child from sibling & rotate keys
- When applicable - preferred



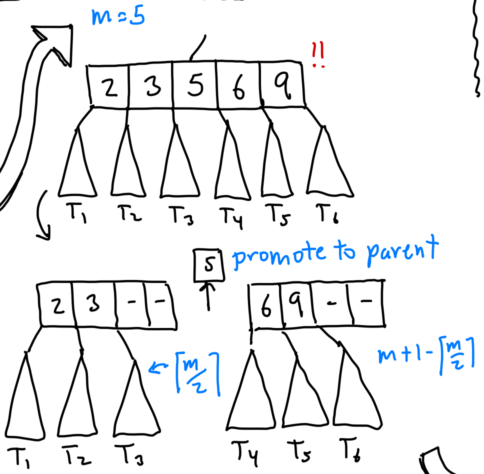
Node Splitting:

- After insertion, a node has too many children... $m+1$
- We split into two nodes of sizes $m' = \lceil m/2 \rceil$ and $m'' = m+1-\lceil m/2 \rceil$

Lemma: For all $m \geq 2$,

$\lceil m/2 \rceil \leq m+1-\lceil m/2 \rceil \leq m$

$\Rightarrow$ $m' + m''$ are valid node sizes



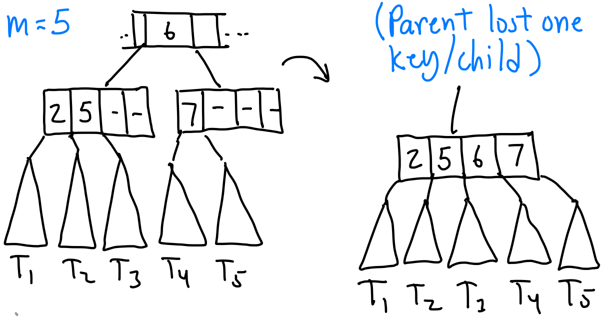
Node Merging:

- A node has too few children $\lceil m/2 \rceil - 1$
- Neither sibling has extra (both $\lceil m/2 \rceil$)
- Merge with either sibling to produce node with $(\lceil m/2 \rceil - 1) + \lceil m/2 \rceil$ child

Lemma: For all $m \geq 2$,

$\lceil m/2 \rceil \leq 2\lceil m/2 \rceil - 1 \leq m$

$\Rightarrow$ Resulting node is valid

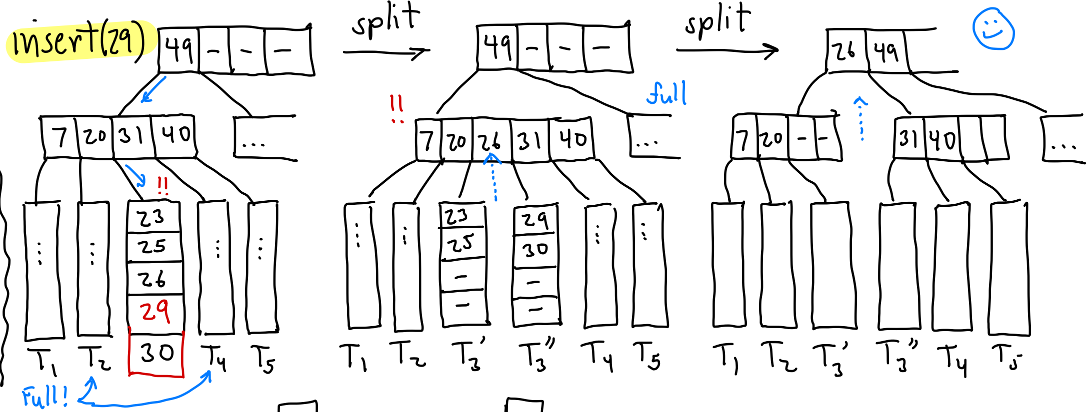


B-Trees III

Insertion:

- Find insertion point (leaf level)
- Add key/value here
- If node overfull (m keys, m+1 children)
  - → Can either sibling take a child (< m)?
  - ⇒ Key rotation [done]
  - → Else, split
    - → Promotes key
    - → If root splits add new root

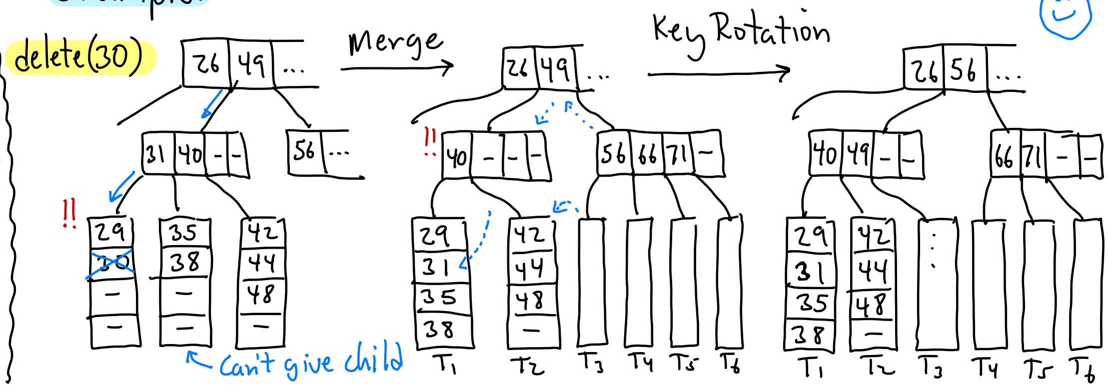
Example: m=5



Deletion:

- Find key to delete
- Find replacement/copy
- If underfull ($\lceil m/2 \rceil - 1$) child
  - → If sibling can give child
    - → Key rotation
  - → Else (sibling has $\lceil m/2 \rceil$)
    - → Merge with sibling
  - → Propagates → If root has 1 child → collapse root

Example: m=5



Scapegoat Trees:

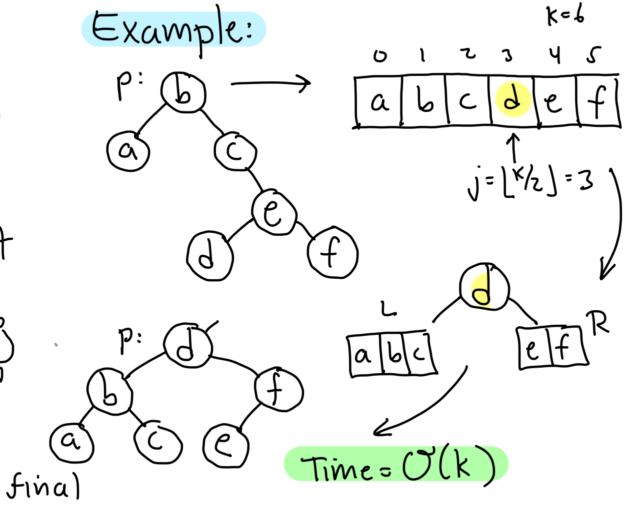
- Arne Anderson (1989)
- Galperin + Rivest (1993) rediscovered/extended
- Amortized analysis
  - O(log n) for dictionary ops amortized (guaranteed for find)
  - Just let things happen
  - If subtree unbalanced – rebuild it

Recap:

- Seen many search trees
- Restructure via rotation
- Today: Restructure via rebuilding
- Sometimes rotation not possible
- Better mem. usage

Scapegoat Trees I

Example:



Time = O(k)

Overview:

Insert:

- same as standard BST
- if depth too high
  - trace search path back
  - find unbalanced node – scapegoat
  - rebuild this subtree

Find:

Same as std BST

- Tree height ≤ $\log_{3/2} n \approx 1.71 \lg n$

Delete:

- Same as std. BST
- If num. of deletes is large rel. to n – rebuild entire tree!

How? Maintain $n, m \leftarrow 0$

Insert: n++, m++

Delete: n-- ⋯→ If $m > 2n$ rebuild

How to rebuild?

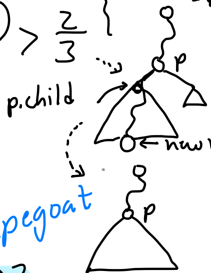
```
rebuild(p):
    - inorder traverse p's subtree → array A[]
    - buildSubtree(A)
buildSubtree(A[0..k-1]):
    - if k=0 return null
    - j ← ⌊k/2⌋; x ← A[j] median
    - L ← buildSubtree(A[0..j-1])
    - R ← buildSubtree(A[j+1..k-1])
    - return Node(x, L, R)
```

Scapegoat Trees II

Insert:

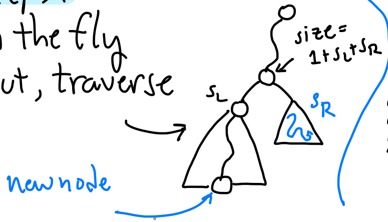
- n++; m++
- same as std BST but keep track of inserted node's depth → d
- if ($d > \log_{3/2} m$) {
  - /* rebuild event */
  - trace path back to root
  - for each node p visited, size(p) = no. of nodes in p's subtree
  - if $\frac{\text{size(p.child)}}{\text{size(p)}} > \frac{2}{3}$

    p ← rebuild(p)
  - break



How to compute size(p)?

- Can compute it on the fly
- While backing out, traverse "other sibling"
- Too slow? No!
  → Charge to rebuild.



Details of Operations:

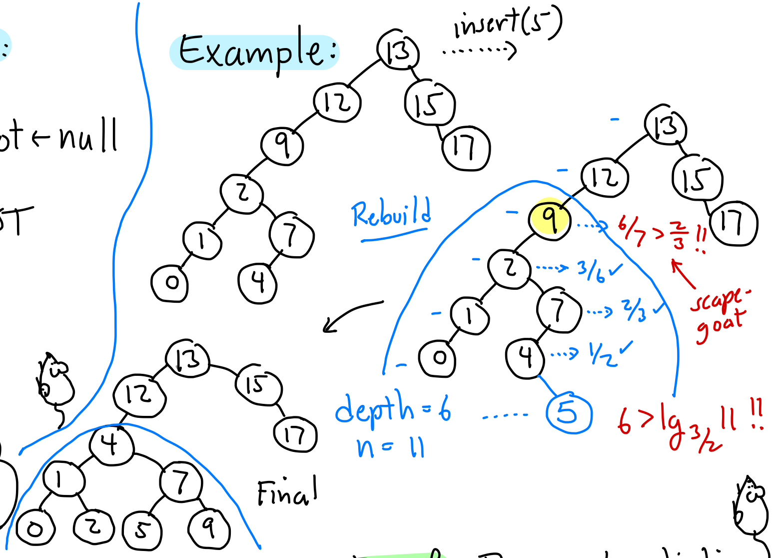
Init: n ← m ← 0 root ← null

Delete:

- Same as std BST
- n--
- if m > 2n, rebuild(root)

Time: O(n)

Example:



Must there be a scapegoat? yes!

Lemma: Given a binary tree with n nodes, if ∃ node p of depth > $\log_{3/2} n$, then ∃ ancestor of p that satisfies scapegoat condition

Proof: By contradiction

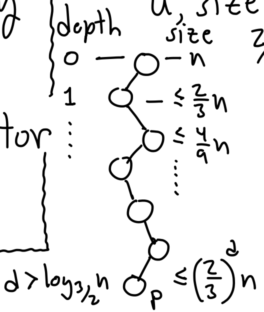
- Suppose p's depth > $\log_{3/2} n$ but ∀ ancestors u, size(u.child) ≤ $\frac{2}{3}$ · size(u)

⇒ Since p has 1 node:

$1 \le \text{size}(p) \le \left(\frac{2}{3}\right)^d n$

$\Rightarrow \left(3/2\right)^d \le n$

$\Rightarrow d \le \log_{3/2} n$ □



Scapegoat Trees

III

Theorem: Starting with an empty tree, any sequence of m dictionary operations on a scapegoat tree take time $O(m \log m)$ [Amortized: $O(\log m)$]

Proof: (Sketch)

Find: $O(\log n)$ guaranteed [Height $= O(\log n)$]

Delete: In order to induce a rebuild, number of deletes ~ number of nodes in tree

→ Amortize rebuild time against delete ops

Insert: Based on potential argument

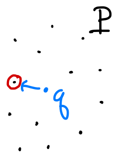
→ It takes ~ k ops to cause a subtree to size k to be unbalanced.

→ Charge rebuild time to these operations

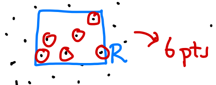
Quadtrees & kd-Trees I

Geometric Search:

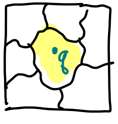
- Nearest neighbors →



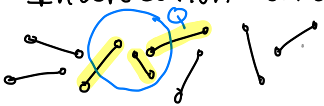
- Range searching



- Point Location



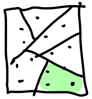
- Intersection Search



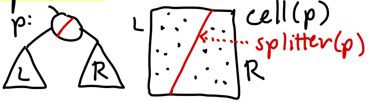
So far: 1-dimensional keys

- Multi-dimensional data
- Applications:
  - Spatial databases + maps
  - Robotics + Auton. Systems
  - Vision/Graphics/Games
  - Machine Learning
- ...

Partition Trees:



- Tree structure based on hierarchical space partition
- Each node is associated w. a region — cell
- Each internal node stores a splitter — subdivides the cell



- External nodes store pts.

Point: A d-vector in $\mathbb{R}^d$

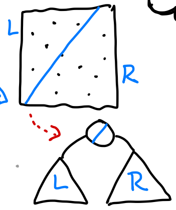
$p = (p_1, \dots, p_d) \quad p_i \in \mathbb{R}$

```
class Point {
   float[] coord  // coords
   Point(int d)
      ....> coord = new float[d]
   int getDim() ..> coord.length
   float get(int i) ..> coord[i]
   .... others: equality, distance
        toString...
}
```

Multi-Dim vs. 1-dim Search?

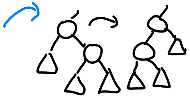
Similarities:

- Tree structure
- Balance $O(\log n)$
- Internal nodes - split
- External nodes - data



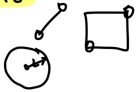
Differences:

- No (natural) total order
- Need other ways to discriminate + separate
- Tree rotation may not be meaningful



Representations:

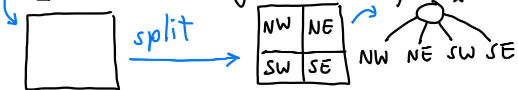
- **Scalars:** Real numbers for coordinates, etc. float
- **Points:** $p = (p_1, \dots, p_d)$ in real d-dim space $\mathbb{R}^d$
- **Other geom objects:** Built from these



Quadtrees & kd-Trees II

Quadtrees: (abstractly)

- Partition trees
- **Cell**: Axis-parallel rectangle
  [AABB - "Axis-aligned bounding box]

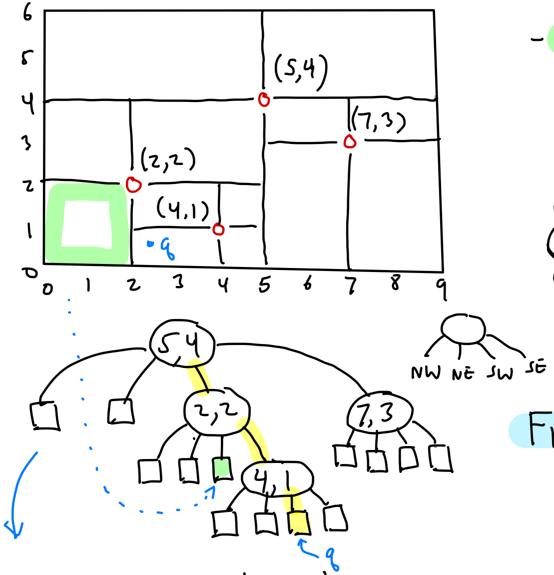


- **Splitter**: Subdivides cell into four (genlly 2^d) subcells

Point Quadtree:

- Each internal node stores a point
- Cell is split by horiz. + vertic. lines through point

(5,4)
(2,2)
(7,3)
(4,1)



Each external node corresps. to cell of final subdivision

Find/Pt Location:

Given a query point q, is it in tree, and if not which leaf cell contains it?

→ Follow path from root down (generalizing BST find)

Quadtrees - Analysis

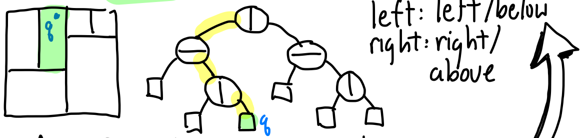
- Numerous variants! PR, PMR, QR, QX, ... see Samet's book
- Popular in 2-d apps (in 3-d, octtrees)
- Don't scale to high dim
  - out degree = 2^d
- What to do for higher dims?

History: Bentley 1975

- called it 2-d tree ($\mathbb{R}^2$)
  3-d tree ($\mathbb{R}^3$)
- In short kd-tree (any dim)
- Where/which direction to split?
  → next

kd-Tree: Binary variant of quadtree

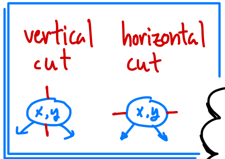
- **splitter**: Horiz. or vertic. line in 2-d (orthogonal plane o.w.)
- **cell**: Still AABB



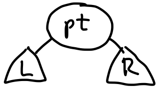
Quadtrees & kd-Trees III

Kd-Tree Node:

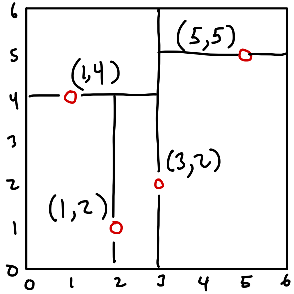
```
class KDNode {
    Point pt     // splitting point
    int cutDim   // cutting coordinate
    KDNode left  // low side
    KDNode right // high side
}
```

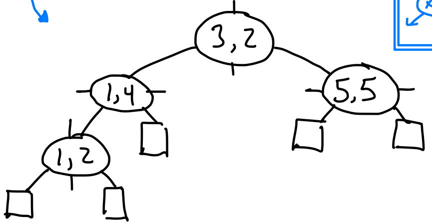


Example:



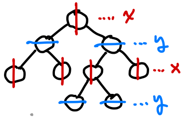
(3,2)
(1,4)
(5,5)
(1,2)



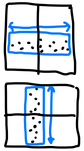


How do we choose cutting dim?

- **Standard kd-tree**: cycle through them (eg. d=3: 1,2,3,1,2,3...) based on tree depth



- **Optimized kd-tree**: (Bentley)
  - Based on widest dimension of pts in cell.



Find point q in subtree

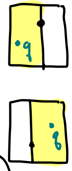
rooted at p with cutDim cd:

- if q == p.point ⇒ found!
- if q[cd] < p.point[cd] ⇒ left
- if q[cd] ≥ p.point[cd] ⇒ right

Helper:

```
class KDNode {
    :
    boolean onLeft(Point q)
    {return q[cutDim] < pt[cutDim]}
}
```

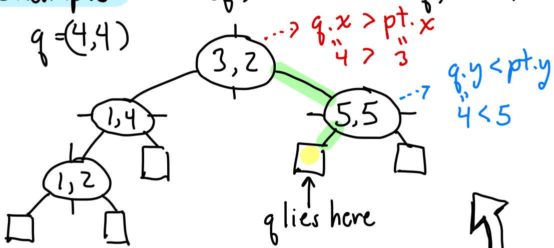
```
Value find(Point q, KDNode p) {
    if (p == null) return null;
    else if (q == p.pt)  ···> all coords match?
        return p.value
    else if (p.onLeft(q))
        return find(q, p.left)
    else
        return find(q, p.right)
}
```



Example:

find(q) $\xrightarrow{\text{calls}}$ find(q, root)

q = (4,4)



Analysis: Find runs in time O(h), where h is height of tree.

Theorem: If pts are inserted in random order, expected height is O(log n)

Quadtrees & kd-Trees IV

Kd-Tree Insertion:

(Similar to std. BSTs)

- Descend tree until
  - → find pt → Error-duplicate
  - → falling out ← (Although we draw extended trees, lets assume standard trees)
    - → create new node
    - → set cutting dim

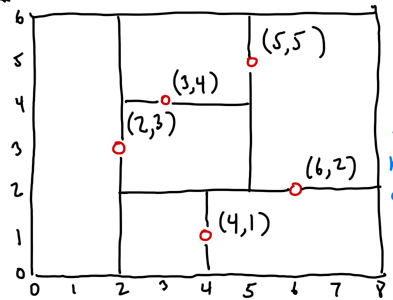
```
KDNode insert(Point pt,
              KDNode p, int cd){
    if(p==null)  //fell out?
        p=new KDNode(pt,cd)
                 //new leaf node
    else if(p.point == pt)
        Error! Duplicate key
    else if(p.onLeft(pt))
        p.left=insert(pt, p.left,(cd+1)%dim)
    else
        p.right=insert(pt,p.right,
                       (cd+1)%dim)
    return p
}
```

cutting dimension to use

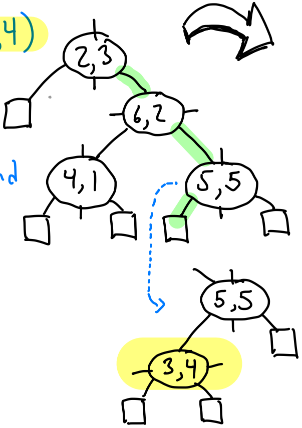
dimension of point

Example:

insert(3,4)



pretend ext. nodes are null



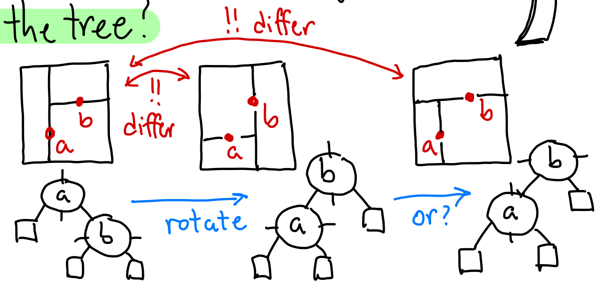
Analysis:

Run time: $O(h)$

Tree height

Can we balance the tree?

- Rotation does not make sense !!



Rebalance by Rebuilding:

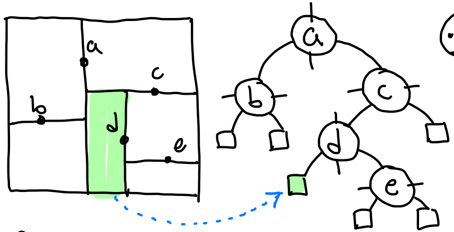
- Rebuild subtrees as with scapegoat trees
- $O(\log n)$ amortized
- Find: $O(\log n)$ guaranteed.

Deletion:

- Descend path to leaf
- If found:
  - leaf node → just remove
  - internal node
    - → find replacement
    - → copy here
    - → recur. delete replacement

This is the hardest part. See Latex notes.

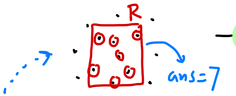
Kd-Tree Queries I

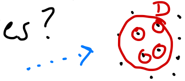


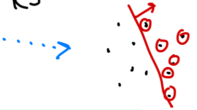
Queries?

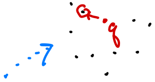
- **Orthogonal range queries**
  - Given query rect. (AABB) count/report pts in this rect.
- Other range queries?
  - Circular disks
  - Halfplane

  ⋮

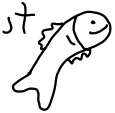








- **Nearest neighbor queries**
  - Given query pt, return closest pt in the set
  - Find k^{th} closest point
  - Find farthest point from q

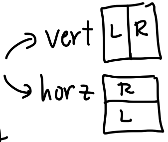


This Lecture: $O(\sqrt{n})$ time alg for orthog. range counting queries in $\mathbb{R}^2$

→ General $\mathbb{R}^d$: $O(n^{1-1/d})$

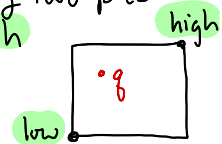
Kd-Trees:

- **Partition trees**
- Orthogonal split
- **Alternate** cutting dimension x, y, x, y, ...
- **Cells** are axis-aligned rectangles (AABB)



Axis-Aligned Rect in $\mathbb{R}^d$

- Defined by two pts: **low**, **high**

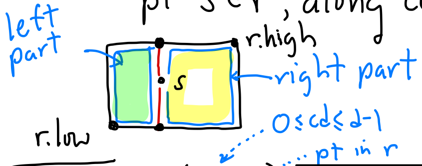


- Contains pt $q \in \mathbb{R}^d$ iff

$low_i \le q_i \le high_i$ $\quad 1 \le i \le d$

Rectangle methods for kd-cells:

- Split a cell r by a split pt $s \in r$, along cutdim cd

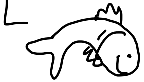


r.leftPart(cd, s)

→ returns rect with low = r.low + high = r.high but high[cd] ← s[cd]

r.rightPart(cd, s)

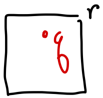
→ high = r.high + low = r.low but low[cd] ← s[cd]



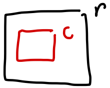
Useful methods:

Let r, c - Rectangle
q - Point

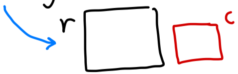
r.contains(q)



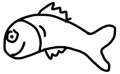
r.contains(c)



r.isDisjointFrom(c)



Orthog. Range Query

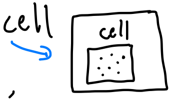


- Assume: Each node p stores:
  - **p.pt** : splitting point
  - **p.cutDim** : cutting dim
  - **p.size** : no. of pts in p's subtree
- Tree stores ptr. to **root** and **bounding box** for all pts.
- Recursive helper stores current **node p** + **p's cell**.

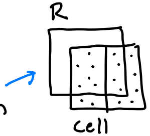
Cases:

- p==null → fell out of tree → 0
- Query rect is **disjoint** from p's cell
  → return 0
  → no point of p contributes to answer

- Query rect **contains** p's cell
  → return p.size
  → every point of p's subtree contributes to answer.

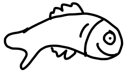


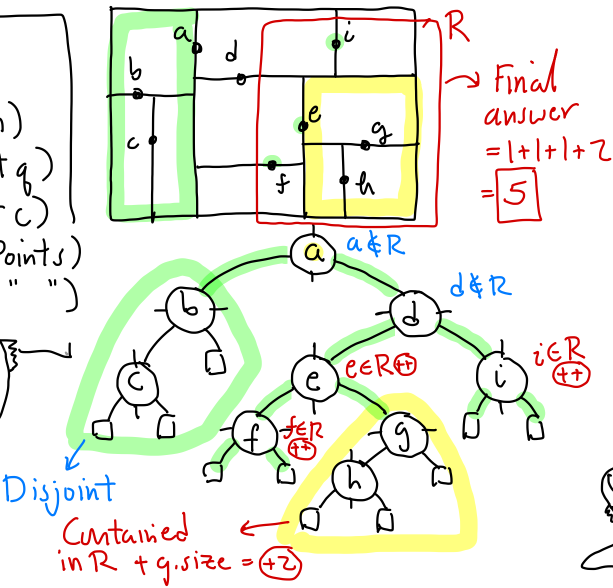
- Otherwise: Rect. + cell **overlap** – Recurse on both children



```
class Rectangle {
  private Point low, high
  public Rect(Point l, Point h)
  "    boolean contains(Point q)
  "    boolean contains(Rect c)
  "    Rect leftPart(int cd, Point s)
  "    Rect rightPart("  "    "  ")
}
```

Kd-Tree Queries II

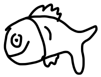




```
int rangeCount(Rect R, KDNode p, Rect cell)
   if (p == null) return 0 // fell out of tree
   else if (R.isDijointFrom(cell)) return 0 // no overlap
   else if (R.contains(cell)) return p.size // take all
   else { int ct = 0
      if (R.contains(p.pt)) ct++ // p's pt in range
      ct += rangeCount(R, p.left,
                    cell.leftPart(p.cutDim, p.pt))
      ct += rangeCount(R, p.right, cell.rightPart...
   }
```

Kd-Tree Queries III

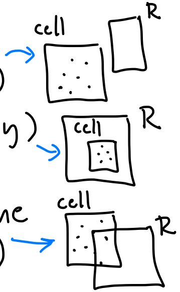
Theorem: Given a balanced kd-tree storing n pts in $\mathbb{R}^2$ (using alternating cut dim), orthog. range queries can be answered in $O(\sqrt{n})$ time.

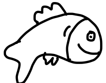


→ Slower than log n. Faster than n

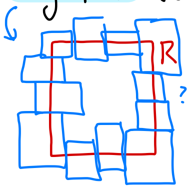
Stabbing: 3 cases

- cell is disjoint (easy)
- cell is contained (easy)
- cell partially overlaps or is stabbed by the query range (hard!)

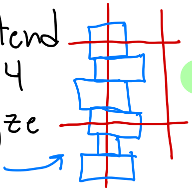




How many cells are stabbed by R? (worst case)



Simpler: Extend R's sides to 4 lines + analyze each one.



Analysis: How efficient is our algorithm?

→ Tricky to analyze

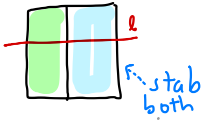
→ At some nodes we recurse on both children ⇒ $O(n)$ time?

→ At some we don't recurse at all!

Lemma: Given a kd-tree (as in Thm above) and horiz. or vert. line ℓ, at most $O(\sqrt{n})$ cells can be stabbed by ℓ

Proof: w.l.o.g. ℓ is horiz.

Cases: p splits vertically



stab both

Solving the Recurrence:

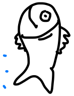
- Macho: Expand it
- Wimpy: Master Thm (CLRS)

Master Thm:

$T(n) = aT(n/b) + n^d$ + $d < \log_b a$

$\Rightarrow T(n) = n^{\log_b a}$

For us: a=2, b=4, d=0 $\Rightarrow T(n) = n^{\log_4 2} = n^{1/2} = \sqrt{n}$



Since tree is balanced a child has half the pts + grandchild has quarter.

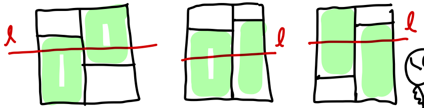
Recurrence: $T(n) = 2 + 2T(n/4)$

2 cells stabbed

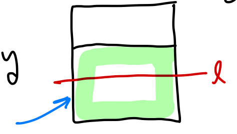
Recurse on 2 grandchildren

Each has n/4 pts

If we consider 2 consecutive levels of kd-tree, ℓ stabs at most 2 of 4 cells:



p splits horizontally



ℓ stabs only one

Hashing I

Recap:
So far, ordered dicts.
- insert, delete, find
- Comparison-based: <, ==, >
- getMin, getMax, getK, findUp...
- Query/Update time: $O(\log n)$
  → Worst-case, amortized, random.

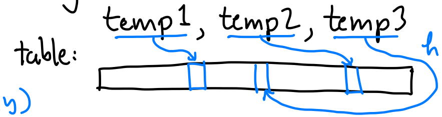
→ Can we do better? $O(1)$?

Hashing: (Unordered)
dictionary
- stores key-value pairs in array table[0..m-1]
- supports basic dict. ops. (insert, delete, find) in $O(1)$ expected time
- does not support ordered ops (getMin, findUp, ...)
- simple, practical, widely used

Overview:
- To store n keys, our table should (ideally) be a bit larger (e.g., $m \geq c \cdot n$, $c = 1.25$)
- Load factor:
  $$\lambda = n/m$$
- Running times increase as $\lambda \to 1$
- Hash function:
  $$h : \text{Keys} \longrightarrow [0..m-1]$$
  → Should scatter keys random.
  → Need to handle collisions → $x \neq y$ but $h(x) = h(y)$

Good Hash Function:
- Efficient to compute
- Produce few collisions
  - Use every bit in key
  - Break up natural clusters

E.g. Java variable names:



Common Examples:
- Division hash:
  $$h(x) = x \bmod m$$
- Multiplicative hash:
  $$h(x) = (ax \bmod p) \bmod m$$
  a, p - large prime numbers
- Linear hash:
  $$h(x) = ((ax + b) \bmod p) \bmod m$$
  a, b, p - large primes

Assume keys can be interpreted as ints

Why "mod p mod m"?
- modding by a large prime scatters keys
- m may not be prime (e.g. power of 2)

Universal Hashing:
Even better → randomize!
- Let H be a family of hash fns
- Select $h \in H$ randomly
- If $x \neq y$ then $\text{Prob}(h(x) = h(y)) = 1/m$

E.g. Let p - large prime, $a \in [1..p-1]$, $b \in [0..p-1]$ all random
- $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$

Hashing II

Overview:

- Separate Chaining
- Open Addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing

simple/slow

complex/fast

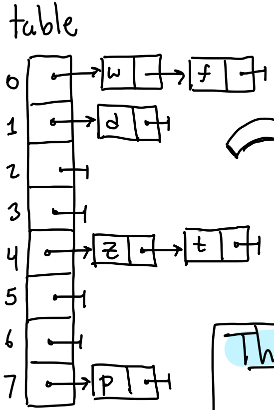
Separate Chaining:

table[i] is head of linked list of keys that hash to i.

Example:

Keys (x)	h(x)
d	1
z	4
p	7
w	0
t	4
f	0

m = 8



Analysis:

Recall load factor

$\lambda = n/m$

n = # of keys

m = table size

Collision Resolution:

If there were no collisions hashing would be trivial!

insert(x,v) → table[h(x)] = v

find(x) → return table[h(x)]

delete(x) → table[h(x)] = null

S_{sc} = Expected search time if x found (successful)

U_{sc} = Expect. search time if x not found (unsuccessful)

Thm: $S_{sc} = 1 + \lambda/2$ $\quad U_{sc} = 1 + \lambda$

Proof: On avg. each list has $n/m = \lambda$

success: 1 for head + half the list

unsuccess: 1 " " + all the list

If $\lambda < \lambda_{min}$ or $\lambda > \lambda_{max}$? Rehash!

- Alloc. new table size = n/λ_0
- Compute new hash fn h
- Copy each x,v from old to new using h
- Delete old table

Token-based – See latex notes!

Thm: Amortized time for rehashing is $1 + (2\lambda_{max}/(\lambda_{max} - \lambda_{min}))$

How to control λ?

- **Rehashing:** If table is too dense / too sparse, realloc. to new table of ideal size

Designer: $\lambda_{min}, \lambda_{max}$ - allowed λ values

$$\lambda_0 = \frac{\lambda_{min} + \lambda_{max}}{2} \text{ "ideal"}$$

If $\lambda < \lambda_{min}$ or $\lambda > \lambda_{max}$ ...

Hashing III

Collision Resolution: (cont.)

- Separate chaining is efficient, but uses extra space (nodes, pointers, ...)
- Can we just use the table itself?

→ **Open Addressing**

Open Addressing:

- Special entry ("empty") means this slot is unoccupied
- Assume $\lambda \leq 1$
- To insert key:

  check: $h(x)$ if not empty try

  $h(x)+i_1$

  $h(x)+i_2$

  $\vdots$

  $\langle i_1, i_2, i_3, \ldots \rangle$ - Probe sequence
- What's the best probe sequence?

Linear Probing:

$h(x), h(x)+1, h(x)+2, \ldots$

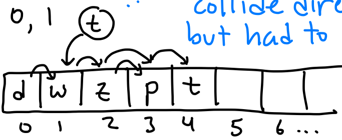
until finding first available

Simple, but is it good?

x: d, z, p, w, t

h(x): 0, 2, 2, 0, 1

table



t did not collide directly but had to probe 3 times!

Analysis:

Let S_{LP} = expected time for successful search

U_{LP} = " " unsuccessful "

Thm:

$$S_{LP} = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$

$$U_{LP} = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)^2$$

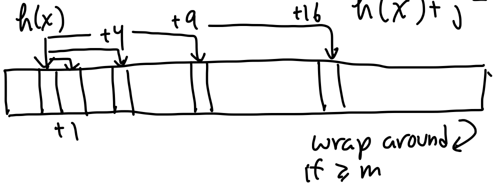
Obs: As $\lambda \to 1$ times increase rapidly

Clustering

- Clusters form when keys are hashed to nearby locations
- Spread them out!

Quadratic Probing:

$h(x), h(x)+1, h(x)+4, h(x)+9, \ldots$ $h(x)+j^2$



Analysis: Improves secondary clustering

- May fail to find empty entry (Try m=4. $j^2 \bmod 4 = 0$ or 1 but not 2 or 3)
- How bad is it? It will succeed if $\lambda < 1/2$.

Thm: If quad. probing used + m is prime, the the first $\lfloor m/2 \rfloor$ probe locations are distinct.

Pf: See latex notes.

Hashing IV

Recap:

Separate Chaining:
Fastest but uses extra space (linked list)

Open Addressing:

- Linear probing:
- Quadratic probing:

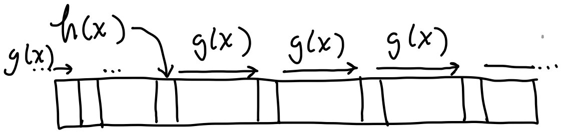
} clustering

Double Hashing:

(Best of the open-addressing methods)

- Probe sequence det'd by second hash fn. - $g(x)$

$$h(x) + \{0, g(x), 2 \cdot g(x), 3 \cdot g(x) \ldots\} \quad [\mathrm{mod}\ m]$$



(until finding an empty slot)

Why does bust up clusters?

Even if $h(x) = h(y)$ [collision] it is very unlikely that $g(x) = g(y)$

⇒ Probe sequences are entirely different!

Analysis: Defs:

S_{DH} = Expected search time of doub. hash. if successful

U_{DH} = Exp. if unsuccessful

Recall: Load factor $\lambda = n/m$

Thm: $S_{DH} = \frac{1}{\lambda} \ln\left(\frac{1}{1-\lambda}\right)$

$U_{DH} = 1/(1-\lambda)$

→ Proof is nontrivial (skip)

λ:	0.5	.075	0.95	0.99
U_{DH}:	2	4	20	100
S_{DH}:	1.39	1.89	3.15	4.65

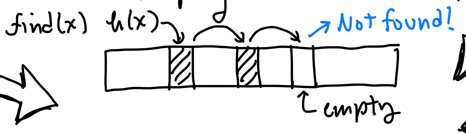
Very efficient!

Dictionary Operations:

Insert(x,v): Apply probe sequence until finding first empty slot.
- Insert (x,v) here.

(If x found along the way ⇒ duplicate key error!)

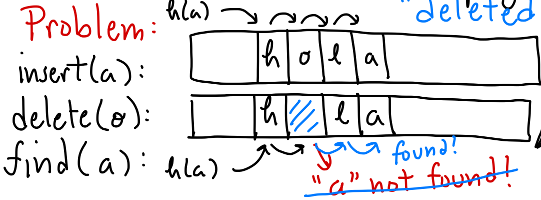
Find(x): Visit entries on probe sequence until:
- found x ⇒ return v
- hit empty ⇒ return null



Delete(x): Apply find(x)
→ Not found ⇒ error
→ Found ⇒ set to "~~empty~~" "deleted"

~~Is this right??~~

Problem:



Range Tree Applications:

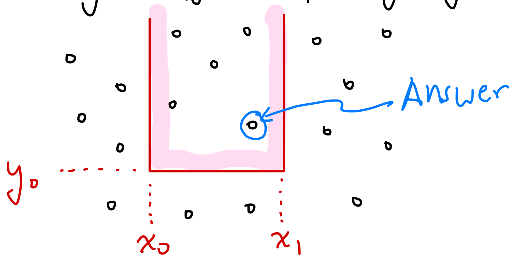
- Range trees can be applied to a variety of query problems

- Methods:
  - Minimization/Maximization
  - Transform coordinates
  - Adding new coordinates

Minimization/Maximization -

3-Sided Min Query

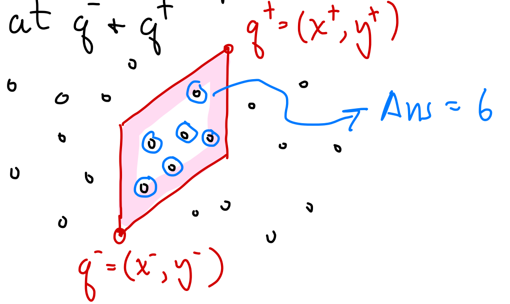
Given a set P of n pts in $\mathbb{R}^2$, a query consists of x-interval $[x_0, x_1]$ and y value y_0. Return the lowest pt in 3-sided region $x_0 \le x \le x_1$ + $y \ge y_0$



Transforming coordinates:

Skewed rectangle query:

Given a set P of n pts in $\mathbb{R}^2$, a skewed rectangle is given by 2 pts $q^- = (x^-, y^-)$ and $q^+ = (x^+, y^+)$ and consists of pts in parallelogram with two vertical sides and two with slope +1 + corners at q^- + q^+

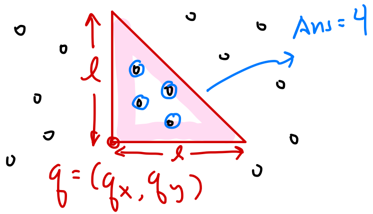


Return a count of the number of pts of P inside the skewed rectangle.

Adding New Coordinates:

NE Right Triangle Query

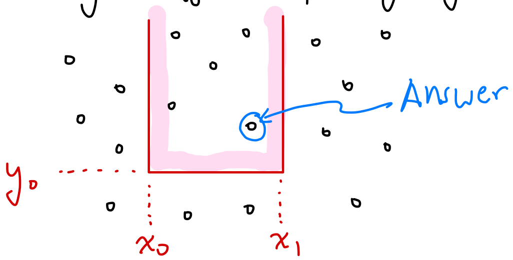
Given a set P of n pts in $\mathbb{R}^2$ and scalar $\ell > 0$, a NE triangle is a 45-45 right triang with lower left corner at q and side length ℓ.



Return a count of the number of pts of P lying within the triangle.

3-Sided Min Query

Return lowest in region
region $x_0 \le x \le x_1$ + $y \ge y_0$



Data structure:

- Build a range tree for x
- Aux. trees are range trees for y that support findLarger

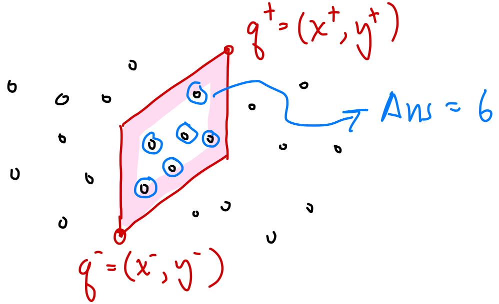
Query Processing:

- Do 1D range search in main tree for interval $[x_0, x_1]$.
- For each maximal subtree in range, do findLarger(y_0)
- Return smallest of these.

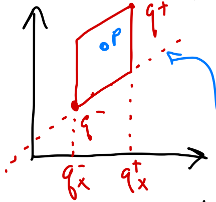
Analysis:

- Same as 2D range tree
- Space: $O(n \log n)$ Time: $O(\log^2 n)$

Skewed rectangle query:



Transform coordinates to make orthog range query



$$q_x^- \le p_x \le q_x^+$$

Line equation:
$y = x + (q_y^- - q_x^-)$

$$p_x + (q_y^- - q_x^-) \le p_y \le p_x + (q_y^+ - q_x^+)$$

$$\Leftrightarrow \quad q_y^- - q_x^- \le p_y - p_x \le q_y^+ - q_x^+$$

$p_y' = p_y - p_x$

Map each $p = (p_x, p_y) \in P$ to $p' = (p_x', p_y') \triangleq (p_x, p_y - p_x)$

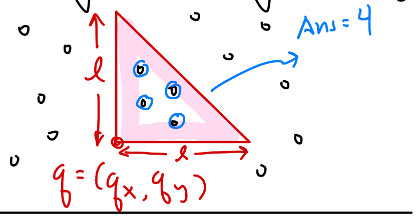
Let P' be resulting set.

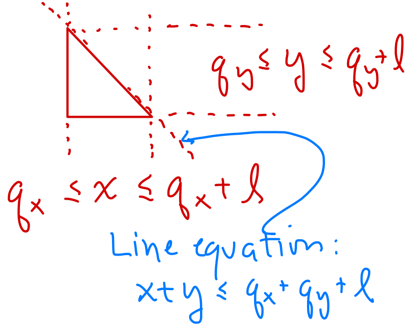
Build std. range tree for P'. Return ans. to query

$$q_x^- \le x \le q_x^+$$

$$q_y^- - q_x^- \le y \le q_y^+ - q_x^+$$

NE Right Triangle Query




$q_y \leq y \leq q_y + \ell$

$q_x \leq x \leq q_x + \ell$

Line equation:
$x + y \leq q_x + q_y + \ell$

- Add new coord:
  $z = x + y$
- Map pts:
  $p = (p_x, p_y) \rightarrow p' = (p_x, p_y, p_x + p_y)$
- Let P' be resulting set

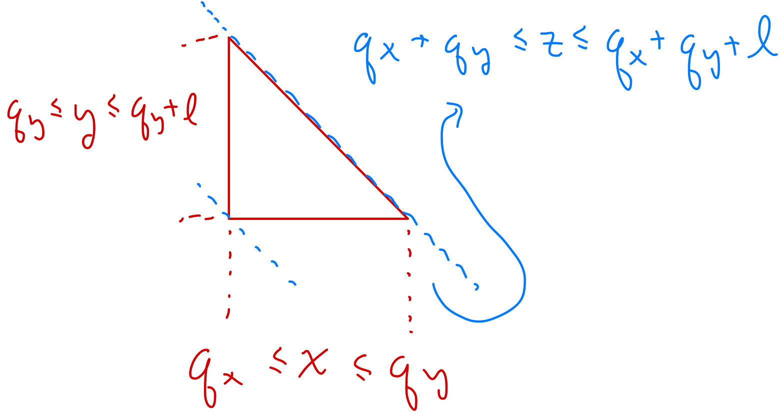
Build a 3D range tree on P'

NE triangle query becomes:

$$q_x \leq x \leq q_x + \ell$$

$$q_y \leq y \leq q_y + \ell$$

$$q_x + q_y \leq z \leq q_x + q_y + \ell$$


$q_x + q_y \leq z \leq q_x + q_y + \ell$

$q_y \leq y \leq q_y + \ell$

$q_x \leq x \leq q_y$

Space:

$O(n \log^2 n)$

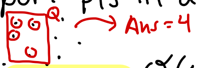
Query time:

$O(\log^3 n)$

Range Trees I

Recap:

- kd-Tree: General-purpose data structure for pts in $\mathbb{R}^d$
- Orthogonal range query: Count/report pts in axis-aligned rect.



- kd-Tree: Counting: $O(\sqrt{n})$ time

  Report: $O(k+\sqrt{n})$ time

  No. of pts reported

Can we do better?

Range Trees:

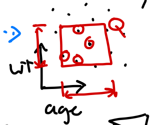
- Space is $O(n \log^{d-1} n)$
- Query time:
  - Counting: $O(\log^d n)$
  - Reporting: $O(k+\log^d n)$

→ In $\mathbb{R}^2$: $\log^2 n$ much better than $\sqrt{n}$ for large n

→ Range trees are more limited

Layering: Combining search structures

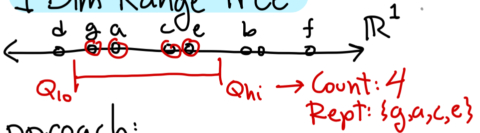
- Suppose you want to answer a composite query w. multiple criteria:
- Medical data: Count subjects w.
  - Age range: $a_{lo} \le age \le a_{hi}$
  - Weight range: $w_{lo} \le weight \le w_{hi}$



- Design a data structure for each criterion individually
- Layer these structures together to answer full query

→ Multi-Layer Data Structures

1-Dim Range Tree:

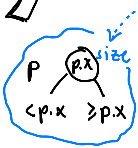


→ Count: 4

Rept: {g,a,c,e}

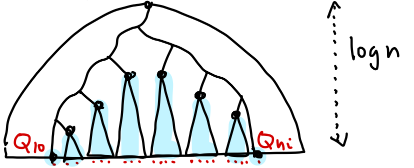
Approach:

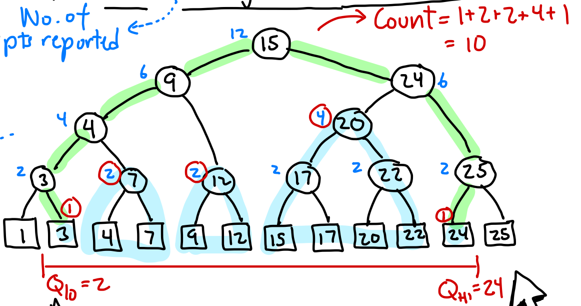
- Balanced BST (eg. AVL, RB, ..)
- Assume extended tree
- Each node p stores no. of entrie in subtree: p.size



Canonical Subsets:

- Goal: Express answer as disjoint union of subsets
- Method: Search for Q_{lo} + Q_{hi} + take maximal subtrees





→ Count = 1+2+2+4+1 = 10

Call this a 1-Dim Range Tree:

Claim: A 1-Dim range tree with n pts has space $O(n)$ and answers 1-D range count/rept queries in time $O(\log n)$ (or $O(k+\log n)$)

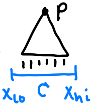
No. of pts reported

Range Trees II

More details:

Given a 1-D range tree T:

- Let $Q = [Q_{lo}, Q_{hi}]$ be **query interval**
- For each node p, define interval **cell** $C = [x_0, x_1]$ s.t. all pts of p's subtree lie in C



- **Root cell**: $C_0 = [-\infty, +\infty]$

Recursive helper:

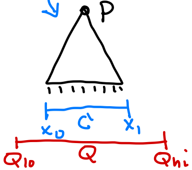
int range1Dx(Node p,
Intv $Q = [Q_{lo}, Q_{hi}]$, Intv $C = [x_0, x_1]$)

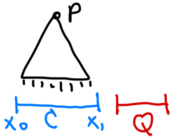
initial call: range1Dx(root, Q, C_0)

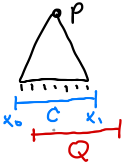
Cases:

p is external:
- if $p.pt.x \in Q \rightarrow 1$ else $\rightarrow 0$

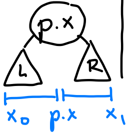
p is internal:
- **$C \subseteq Q$** $\Rightarrow$ all of p's pts lie within query
  → return **p.size**







- **C is disjoint from Q** $\Rightarrow$ none of p's pts lie in Q
  → return **0**
- Else **partial overlap**
  → Recurse on p's children + trim the cell



```
int range1Dx(Node p,
    Intv Q, Intv C=[x_0,x_1]) {
    if(p is external)          T -> 1
        return p.pt.x ∈ Q      F -> 0
    else if (C ⊆ Q) return p.size
    else if (Q+C disjoint) return 0
    else return:
        range1Dx(p.left, Q, [x_0, p.x])
        + range1Dx(p.right, Q, [p.x, x_1])
```

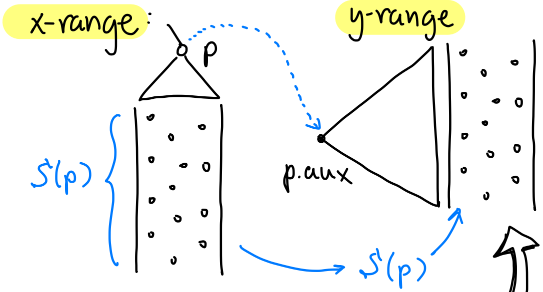
Analysis:

Lemma: Given a 1-D range tree with n pts, given any interval Q, can compute $O(\log n)$ subtrees whose union is answer to query.

Thm: Given 1-D range tree... can answer range queries in time $O(\log n)$ ····→ (+k to report)

2-D Range Searching:

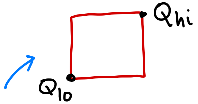
- "**Layer**" a range tree for x with range tree for y
- For each node $p \in$ 1D-x tree, let **$S(p)$** = set of pts in p's subtree
- Def: **p.aux**: A 1D-y tree for $S(p)$



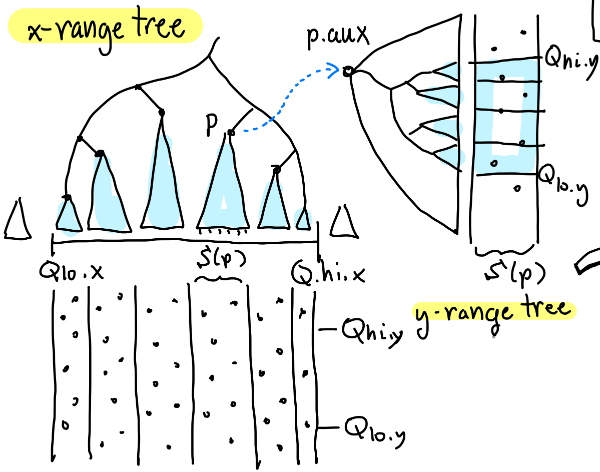
Range Trees III

Answering Queries?

Given query range $Q=[Q_{lo}.x, Q_{hi}.x]\times[Q_{lo}.y, Q_{hi}.y]$



- Run range1Dx to find all subtrees that contribute
- For each such node p, run range1D$_y$ on p.aux
- Return sum of all result



Intuition: The x-layer finds subtrees p contained in x-range + each aux tree filters based on y.

2D Range Tree:

- Construct 1D range tree based on x coords for all pts
- For each node p:
  - Let S(p) be pts of p's tree
  - Build 1D range tree for S(p) based on y → p.aux
- Final structure is union of x-tree + (n-1) y-trees

Higher Dimensions?

- In d-dim space, we create d-layers
- Each recurses one dim lower until we reach 1-d search
- Time is the product:

$$\underbrace{\log n \cdot \log n \cdots \log n}_{d} = O(\log^d n)$$

Analysis: The 1D x search takes of $O(\log n)$ time + generates $O(\log n)$ calls to 1D y search

$\Rightarrow$ Total: $O(\log n \cdot \log n) = O(\log^2 n)$

```
int range2D(Node p, Rect Q, Intv $C=[x_0,x_1]$){
    if (p is external) return p.pt ∈ Q? $\xrightarrow{T}$ 1 $\xrightarrow{F}$ 0
    else if (Q.x contains C){          // $C \subseteq$ Q's x-projection
        $[y_0,y_1] = [-\infty,+\infty]$          // init y-cell
        return range1Dy(p.aux, Q, $[y_0,y_1]$)
    } else if (Q.x is disjoint of C) return 0
    else                               // partial x-overlap
        return range2D(p.left, Q, $[x_0, p.x]$)
            + range2D(p.right, Q, $[p.x, x_1]$)
}
```

Analysis:

Invoked $O(\log n)$ times - once per maximal subtree

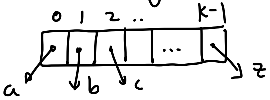
Invoked $O(\log n)$ times - once for each ancestor of max subtree

Tries and Digital Search Trees I

Tries: History

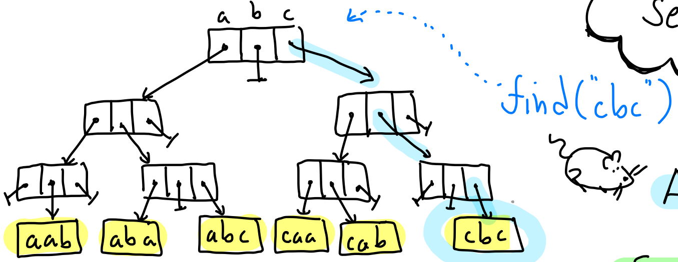
- de la Briandais (1959)
- Fredkin - "trie" from "retrieval"
- Pronounced like "try"

Node: Multiway of order k

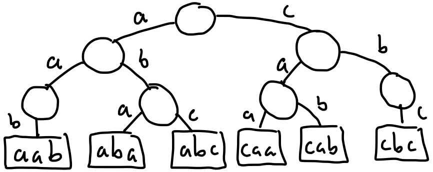


Example: $\Sigma = \{a=0, b=1, c=2\}$

Keys: {aab, aba, abc, caa, cab, cbc}



Same structure/Alt. Drawing



Digital Search:

- Keys are strings over some alphabet Σ
- E.g. $\Sigma = \{a, b, c, \dots\}$
  $\Sigma = \{0, 1\}$ Let $k = |\Sigma|$
- Assume chars coded as ints: $a=0, b=1, \dots z=k-1$

Analysis:

Search: ~ length of query string [O(1) time per node]

Space:

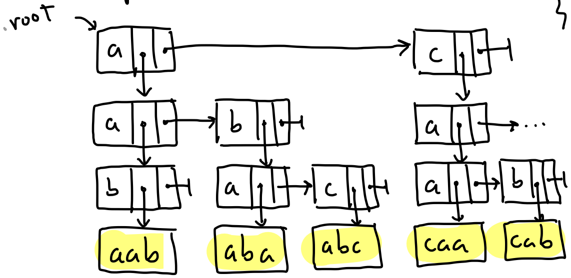
- No. of nodes ~ total no. of chars in all strings
- Space ~ k·(no. of nodes)

Large!

Analysis:

- **Space**: Smaller by factor k
- **Search Time**: Larger by factor of k

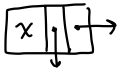
Example:



How to save space?

de la Briandais trees:

- Store 1 char. per node
-



≠ x ⇒ try next char in Σ

== x ⇒ advance to next character of search string

- First-child/next-sibling

Tries and Digital Search Trees II

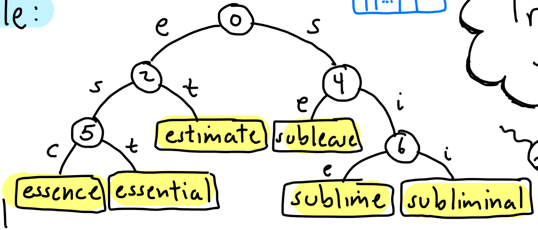
Patricia Tries:

- Improves trie by compressing degenerate paths
- PATRICIA = Practical Alg. to Retrieve Info. Coded in Alpha...
- Late 1960's: Morrison + Gwehenberger
- Each node has index field, indicates which char to check next (Increase with depth)

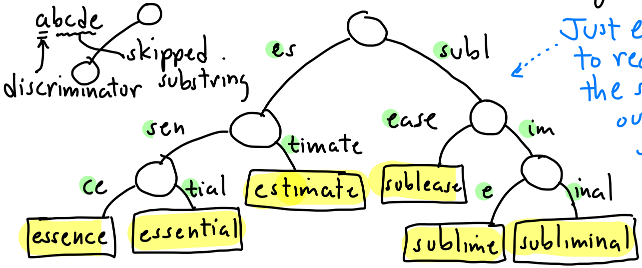
Branch based on ith char of string

Example:

0 1 2 3 ...
essence
essential
estimate
sublease
sublime
subliminal



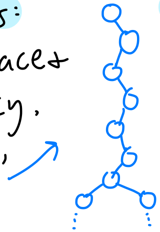
Same data structure – Drawn differently



Just easier to read the strings out.... same data struct.

Dealing with long Paths:

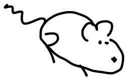
- To get both good space + query time efficiency, need to avoid long, degenerate paths.



- Path compression!

Analysis:

- Query time: (Same as std trie) ~ search string length (may be less)
- Space:
  - No. nodes: ~ No. of strings (irresp. of length)
  - Total space: $K \cdot$ (No. of nodes) + (Storage for strings)



Example:

	ID		ID
S_{10} : \$	\$	S_5 : ajam...	aj
S_9 : a\$	a\$	S_4 : pajam...	paj
S_8 : ma\$	ma\$	S_3 : apaja...	ap
S_7 : ama\$	ama\$	S_2 : mapaj...	map
S_6 : jama\$	j	S_1 : amapaj...	amap
		S_0 : pamapa...	pam

Example: S = pamapajama\$

0 1 2 3 4 5 6 7 8 9 10

S_{10} - \$
S_9 - a\$
S_8 - ma\$
S_7 - ama\$
⋮

Def: Substring identifier for S_i is shortest prefix of S_i unique to this string

E.g. ID(S_1) = "amap"
ID(S_7) = "ama\$"

Suffix Trees:

- Given single large text S
- Substring queries: "How many occurrences of "tree" in CMSC 420 notes"

Notation: $S = a_0 a_1 a_2 \ldots a_{n-1}\$$ special terminal

- Suffix: $S_i = a_i a_{i+1} \ldots a_{n-1}\$$
- Q: What is minimum substring needed to identify suffix S_i?

Tries and Digital Search Trees III

Suffix Trees (cont.)

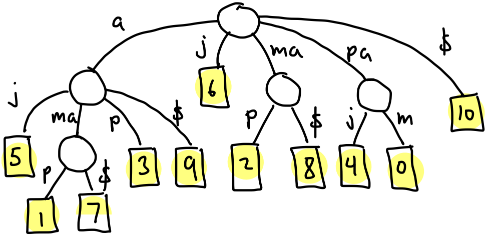
S - text string $|S| = n$

S_i = i^{th} suffix

Substring ID = min substr. needed to identify S_i

A **suffix tree** is a Patricia trie of the n+1 substring identifiers

Example: $S = \overset{0}{p}\overset{1}{a}\overset{2}{m}\overset{3}{a}\overset{4}{p}\overset{5}{a}\overset{6}{j}\overset{7}{a}\overset{8}{m}\overset{9}{a}\overset{10}{\$}$



E.g. ID(S_1) = amap ID(S_7) = ama$

Substring Queries:

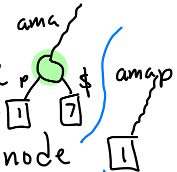
How many occurrences of t in text?

- Search for target string t in trie
- if we end in internal node (or midway on edge) - return no. of extern. nodes in this subtree
- else (fall out at extern. node)
  - compare target with string
    - if matches - found 1 occurrence
    - else - no occurrences

Example:

Search("ama") → End at intern node

Report: 2 occ's.



Search("amapaj") → End at extern node

Go to S_1 + verify

Analysis:

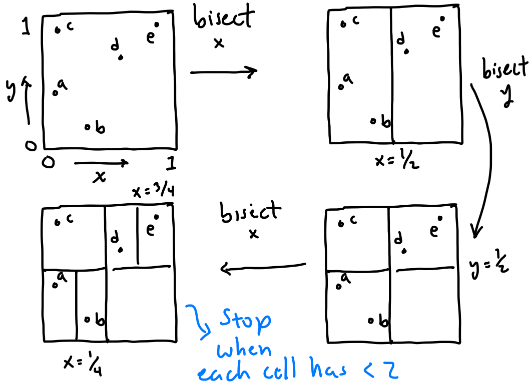
- **Space:** O(n) nodes
  O(n·k) total space
  (k = |Σ| = O(1))
- **Search time:** ~ total length of target string
- **Construction time:**
  - O(n·k) [nontrivial]

PR k-d tree: Can be used for answering same queries as point kd-tree (orth. range, near. neigh)

Geometric Applications:

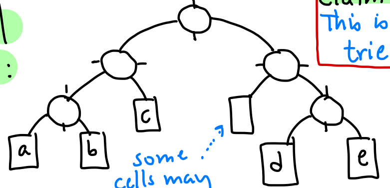
PR kd-Tree: kd-tree based on midpoint subdivision

Assume points lie in unit square



Stop when each cell has < 2 points

Final tree:



some cells may be empty

Claim: This is a trie!

Tries and Digital Search Trees IV

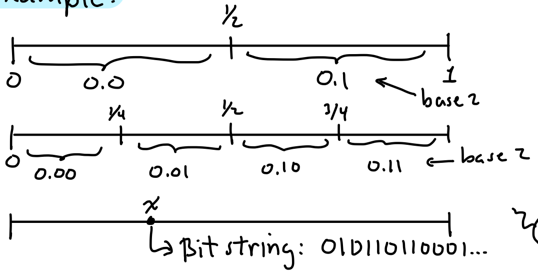
Binary Encoding:

- Assume our points are scaled to lie in unit square $0 \le x, y < 1$ (can always be done)
- Represent each coordinate as binary fraction:

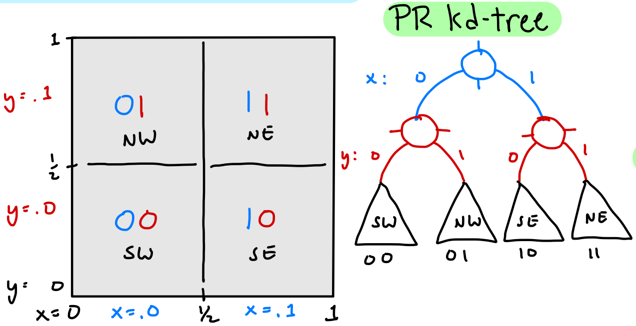
$$x = 0.a_1a_2a_3\dots \qquad a_i \in \{0,1\}$$

$$x = \sum a_i \cdot 1/2^i$$

Example:



How do we extend to 2-D?



PR kd-Tree ≡ Trie ??

- Approach: Show how to map any point in $\mathbb{R}^2$ to bit string
- Store bit strings in a trie (alphabet $\Sigma = \{0,1\}$)
- Prove that this trie has same structure as kd-tree

Bit Interleaving:

Given a point $p = (x, y)$

$0 \le x, y < 1$

let: $x = 0.a_1a_2\dots$ in binary

$y = 0.b_1b_2\dots$

Define:

$$\phi(x,y) = a_1b_1a_2b_2a_3b_3\dots$$

Called **Morton Code** of p

Further Remarks:

- Techniques for efficiently encoding, building, serializing, compressing... tries apply immediately to PR kd-tree
- Can generalize to any dimension

$$\left.\begin{array}{l} x = 0.a_1a_2\dots \\ y = 0.b_1b_2\dots \\ z = 0.c_1c_2\dots \end{array}\right\} \phi = a_1b_1c_1a_2b_2c_2\dots$$

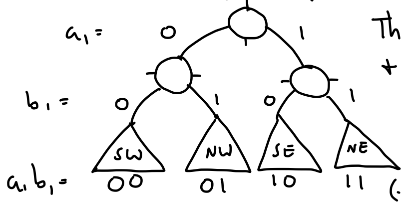
Lemma: Given a pt set $P \subseteq \mathbb{R}^2$ (in unit square $[0,1)^2$) let $P = \{p_1, \dots, p_n\}$ where $p_i = (x_i, y_i)$

Let $\Phi(P) = \{\phi(p_1), \phi(p_2), \dots, \phi(p_n)\}$ (n binary strings)

Then the PR kd-tree for P is equivalent to binary trie for $\Phi(P)$.

Proof: By induction on no. of bits

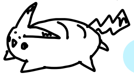
Let $x = 0.a_1a_2\dots \quad y = 0.b_1b_2\dots$ and consider just $\phi(x,y) = a_1b_1\dots$



The PR kd-tree + binary trie assign pts to same subtrees.

(.... induction)

Memory Management I



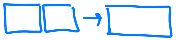
Deallocation Models:

Explicit: (C + C++)
- programmer deletes
- may result in leaks if not careful

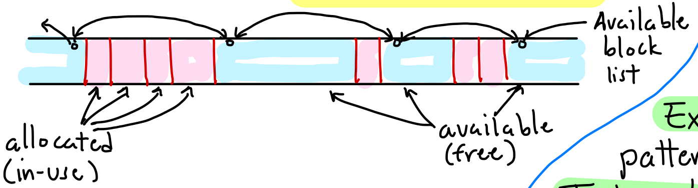
Implicit: (Java, Python)
- runtime system deletes
  - Garbage collection
  - Slower runtime
  - Better memory compaction

Explicit Allocation/Deallocation

- Heap memory is split into blocks whenever requests made
- Available blocks:
  - merged when contiguous



  - stored in available block list



What happens when you do

- new (Java)
- malloc/free (C)
- new/delete (C++) ?

Runtime System Mem. Mgr.

- Stack - local vars, recursion
- Heap - for "new" objects

Don't confuse with heap data structure / heapsort

Fragmentation:

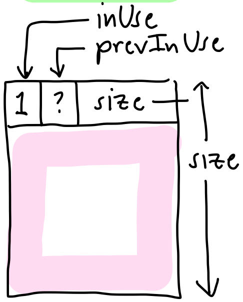
- Results from repeated allocation + deallocation (Swiss-cheese effect)

External: Caused by pattern of alloc/dealloc

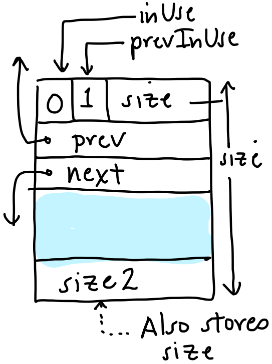
Internal: Induced by mem. manage. policies (not user)

Block Structure:

Allocated:



Available:



Guide:

prevInUse: 1 if prev. contig. block is allocated

prev/next: links in avail. list

size/size2: total block size (includes headers)

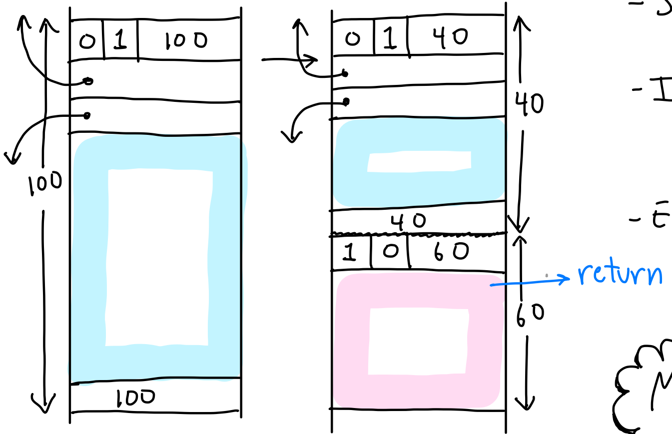
How to Select from available blocks?

- **First-fit:** Take first block from avail. list that is large enough
- **Best fit:** Find closest fit from avail list

Surprise: First-fit is usually better
- faster + avoids small fragments

Memory Management II

Example: Alloc b=59



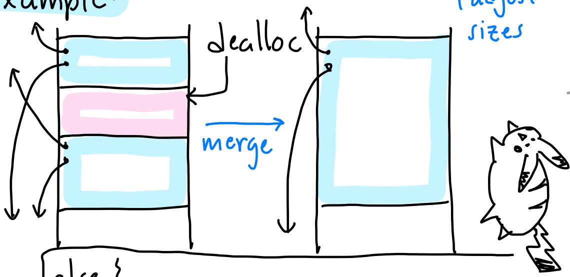
Allocation: malloc(b)

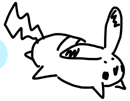
- Search avail. list for block of size b' ≥ b+1
- If b' close to b: alloc entire block (unlink from avail list)
- Else: Split block

Deallocation:

- If prev + next contiguous blocks are allocated → add this to avail
- Else - merge with either/both to make max. avail block

Example:





Some C-style pointer notation

void* - pointer to generic word of memory

Let p be of type void*:

p+10 - 10 words beyond p

*(p+10) - contents of this

Let p point to head of block:

p.inUse, p.prevInUse, p.size

- we omit bit manipulation

*(p+p.size-1) - references last word in this block

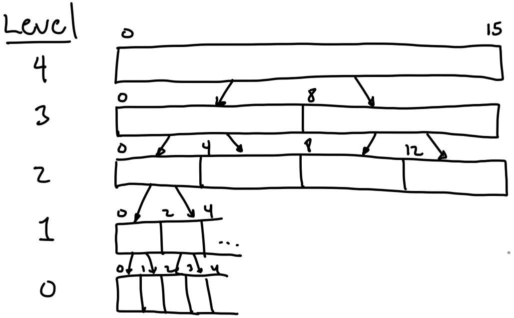
```
(void*) alloc(int b) {
    b += 1               // add +1 for header
    p = search avail list for block
            size ≥ b
    if (p == null) Error-Out of mem!
    if (p.size - b ≤ TOO_SMALL)
        unlink p from avail. list
        q = p
    else .... (continued)

    else {
        p.size -= b                 // remove allocation
        *(p+p.size-1) = p.size      // size 2
        q = p + p.size              // start of new block
        q.size = b                  // new block
        q.prevInUse = 0             //   header
    }
    q.inUse = 1
    (q + q.size).prevInUse = 1
            // update prevInUse for next contig. block
    return q+1                      // skip over header
}
```

Memory Management III

Buddy System:

- Block sizes (including headers) are power of 2
- Requests are rounded up (internal fragmentation)
- Block size 2^k starts at address that is multiple of 2^k
- k = level of a block

Structure:

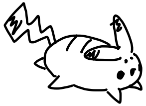


In practice: There is a minimum allowed block size

Buddy system only allows allocations aligning with these blocks

Coping with External Fragmentation

- Unstructured allocation can result in severe external fragmentation
- Can we compress? Problem of pointers
- By adding more structure we can reduce extern frag. at cost of internal frag.



Merging:

- When two adjacent blocks are available, we don't always merge them

→ Must have same size: 2^k

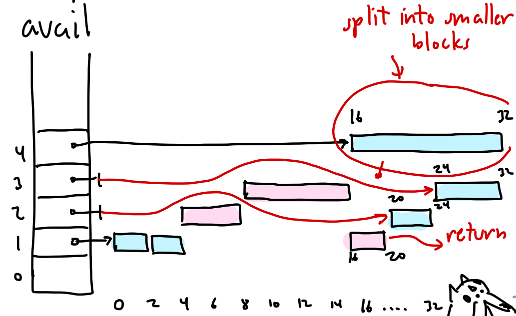
→ Must be buddies - siblings in this tree structure

Def: $\text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } 2^{k+1} \text{ divides } x \\ x - 2^k & \text{otherwise} \end{cases}$

$\equiv \text{buddy}_k(x) = (1 \ll k) \oplus x$ [Bit manipulation]

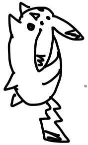
Example:

$\text{alloc}(2) \xrightarrow{\text{round up}} \text{alloc}(4)$



Allocation: alloc(b)

- $k = \lceil \lg(b+1) \rceil$ add +1 for header
- if avail[k] non empty - return entry + delete
- else: find avail[j] ≠ ∅ for j > k
  - split this block



Big Picture:

- Avail list is organized by level: avail[k]
- Block header structure same as before except:

prevInUse } not needed
size2

Quake Heaps I

- Basic definitions
- Operations

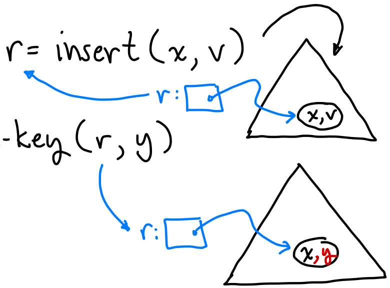
Heap: Review

- A data structure storing key-value pairs
- Supports (at a minimum)
  - insert(Key x, Value v)
  - extract-min()
- Example: Binary heap used in Heapsort

Decrease-Key:

- Given an entry (x,v), decrease the key value from x to y
- How to identify the entry?
  - Heaps do not support an efficient way to find keys
- Locator: A special (abstract) object that identifies an entry of the heap.

Locator r= insert(x, v)
:
decrease-key(r, y)



- Why not just return a pointer to node (x,v)? Private information
  - Locator is a public object (eg. an inner class of the Heap)
- How about increase-key?
  - Heaps are very asymmetrical w.r.t. keys

Why decrease-key?

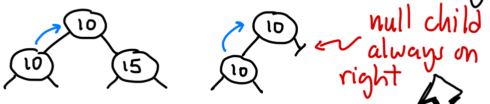
- Dijkstra's algorithm
- Heap tracks distances to vertices from source
- n extract-mins
- upto n^2 decrease-keys
- want decrease key fast!

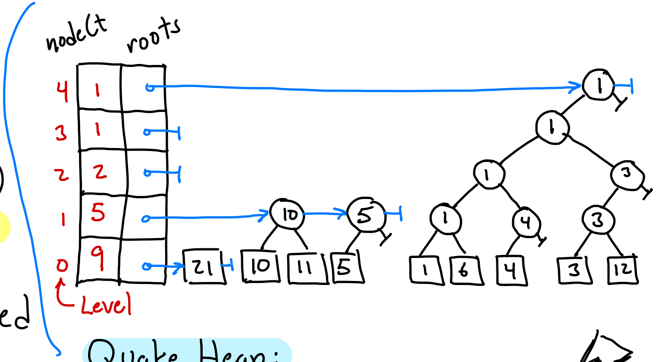
History:

1984: Fibonacci Heaps
(Fredman + Tarjan) ← Complex to analyze
⋮ many variants
2013: Quake Heap ← Much simpler
(Timothy Chan)

Quake Heap:

- Collection of binary trees
- Nodes organized in levels
- All entries are leaves at level 0
- Internal nodes have 1 or 2 children
- Parent stores smaller of child keys





Quake Heaps II

- Utility ops
- Insert
- Decrease-key

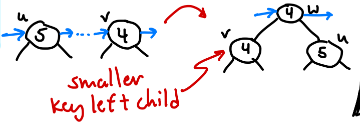
Basic utilities:

make-root(Node u): Make u a root

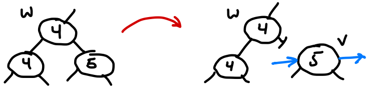
trivial-tree(Key x): Create 1-node tree with key x

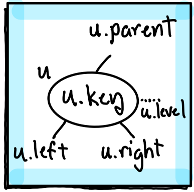
link(Node u, Node v): Link u + v

- u+v roots on same level



cut(Node w): Assuming w has right child – cuts it off as new root





void make-root(Node u)

```
u.parent ← null
add u to roots[u.level]
```

Node trivial-tree(Key x)

```
Node u ← new Node key x + level 0
nodeCt[0] += 1
make-root(u)
return u
```

Node link(Node u, Node v)

```
int lev ← u.level+1   (== v.level+1)
if(u.key ≤ v.key)
        w ← new Node(u.key, lev, u, v)
                             left child  right child
else  w ← new Node(v.key, lev, v, u)
nodeCt[lev] += 1
u.parent ← v.parent ← w
return w
```

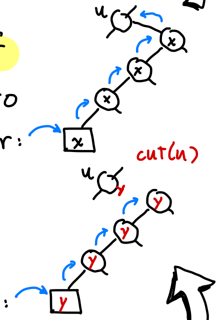
void cut(Node w)

```
Node v ← w.right
if (v ≠ null)
    w.right ← null
    make-root(v)
```

We'll apply these utilities to implement operations

Decrease Key:

- Use locator to access leaf
- Follow left-child path to highest ancestor
- Cut(u): Now we are free to change key
- In code, we'll change up order of ops



void decrease-key(Locator r, Key y)

```
Node u ← r.getNode()   // get leaf node
Node uChild ← null
do {
    u.key ← y   // update key value
    uChild ← u; u ← u.parent   // go up
} while (u ≠ null && uChild == u.left)
if (u ≠ null) cut(u)   // cut subtree
```

Insert:

Super lazy! Just make a single node tree

Locator insert(Key x)

```
Node u ← new trivial-tree(x)
return new Locator(x)
```

Quake Heap III

- Extract Min

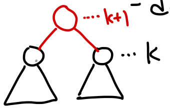
So far:

- insert + decrease-key - lazy!
  - Don't worry about
    - tree balance
    - number of roots
- insert - $O(1)$ time
- dec-key - $O(\log n)$ [later: $O(1)$]

Extract-Min:

- Find the root with smallest key (brute force)
- Delete all nodes down to leaf - many trees

- Merge trees together
  - Work bottom-up
  - Merge 2 trees at level k to form tree at lev k+1



- Too "stringy"? → Flatten QUAKE!

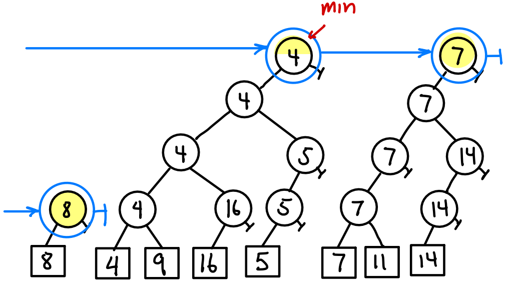
Quake:

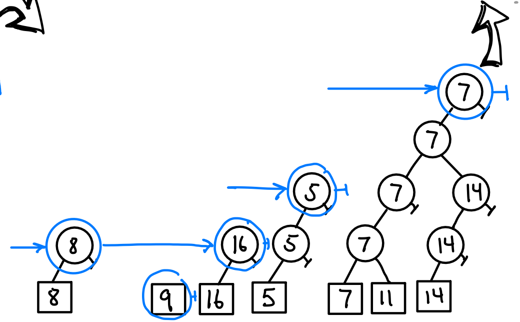
```
for (k=0,1,2,...,nLevels-2) {
    if (nodeCt[k+1] > 0.75 * nodeCt[k])
        - remove all nodes at level k+1 and higher
        - make all nodes at level k roots
```

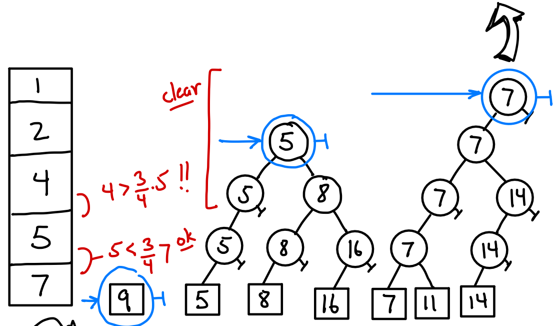
Intuition: Tree becomes "stringy" after many extractions.

- This is evidenced by the fact that node counts do not decrease
- When this happens - we flatten so we can build up later

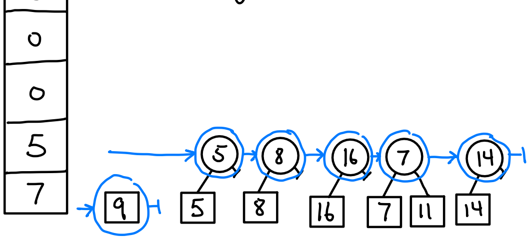
Extract Min Example:







Finally, return 4



Quake Heaps IV

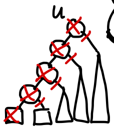
- Extract min (cont)
- Faster decrease key

Key extract-min()

```
Node u ← find root (all levels)
        with smallest key
Key result ← u.key
delete-left-path(u)
remove u from roots[u.level]
merge-trees()
quake()
return result
```

void delete-left-path(u)

```
while (u ≠ null)
    cut(u)
    nodeCt[u.level] -= 1
    u ← u.left
```



void merge-trees()

```
for (lev ← 0.. nLevels-2)
    while (roots[lev].size >= 2)
        Node u,v ← remove any 2
                    from roots[lev]
        make-root(link(u,v))
```

Extract-min: Recap

- find root with min key
- delete left-chain to leaf
- merge trees
- quake (if needed)
- return result

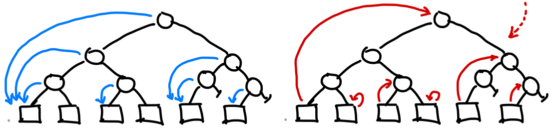
void quake()

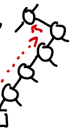
```
for (lev ← 0.. nLevels-2)
    if (nodeCt[lev+1] > 3/4·nodeCt[lev])
        clear-all-above(lev)
```

Clear-all-above (lev) removes all nodes in levels lev+1 .. nLevels-1 and makes nodes of lev into roots

Faster Decrease-key:

- Each node stores pointer to leaf with key (only one change)
- Each leaf stores highest left chain ancestor (path trace O(1) time)





Decrease-key:

- Locate leaf node - O(1)
- Trace path up left-child links
- Cut O(1)
- Change key

O(height) = O(log n)

Times:

Insert - O(1)

Decrease-key - O(log n)

Extract-min - ??

Can we do better? O(1)?

will show O(log n) amortized

Quake Heaps V

- Analysis
  (Quick + Dirty)

Why is Quake Heap efficient?

- **insert**: $O(1)$ worst case
- **decrease-key**: $O(1)$ worst case
  (assuming enhancements)
- **extract-min**: As bad as
  $O(n)$ [no. of roots]

Amortized Analysis:

- Can show that extract-min runs in $O(\log n)$ **amortized time**
- Given any sequence of ops (starting from empty heap) **time to do m ops** (insert, dec-key, extract-min) is
  $O(m \cdot \log n)$
  n = max no. of keys

Potential-Based Analysis:

- Each instance of the data structure assigned a **potential** Ψ

- **Low potential** ⇒ **good** structure
- **High potential** ⇒ **bad** structure

Idea: The **amortized cost** of an operation defined to be
(actual-cost) + (change in Ψ)

Intuition: Expensive ops okay if they **improve structure**
actual = high $\Delta\Psi$ = negative

Quake Heap Potential:

Let **N** = no. of nodes
R = no. of roots
B = no. of nodes with 1 child (bad nodes)

$$\Psi = N + 2R + 4B$$

Lemma: Amortized cost of
insert / dec-key = $O(1)$
extract-min = $O(\log n)$

Intuition:

Extract min actual cost is high ⇒
- **Tree height > $O(\log n)$**
  - Quake will flatten
- **Many more roots than $O(\log n)$**
  - Merge trees will reduce no. to $O(\log n)$

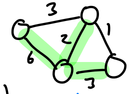
Potential decrease compensates for high actual cost

Data Structures & Algorithm Design: Euclidean Min. Spanning Tree (I)

Minimum Spanning Trees:

- Given a connected, weighted graph $G=(V,E)$

  $(u,v)\in E \rightarrow w(u,v) = \text{weight}$

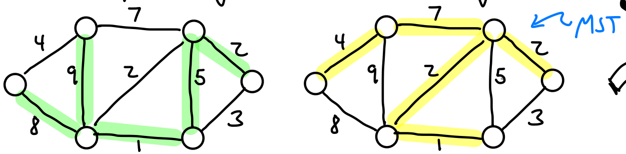


- **Spanning Tree:**
  - A subset $T \subseteq E$ of edges that connect all the vertices and is acyclic

Total weight: $w(T)=\sum_{(u,v)\in T} w(u,v)$

Minimum Spanning Tree (MST)

- A spanning tree of min. weight



$w(T)=8+9+2+\ldots=25$ $w(T)=1+2+2+7+4=16$

Facts:

- If G has n vertices, any spanning tree has n-1 edges

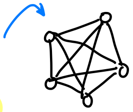
How are data structures used?

- Transaction/Query:
  - Insert new student name="Mary" ID=1234...
  - Closest coffee to my location
- Algorithms:
  - Dijkstra – Fibonacci Heap
  - Kruskal – Union/Find

Euclidean Graph:

Given a set $P=\{p_1,\ldots,p_n\}$ of pts in $\mathbb{R}^d$, this is a **complete graph** (all $\binom{n}{2}$ edges) where:

$$w(p_i,p_j)=\text{dist}(p_i,p_j)=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$$

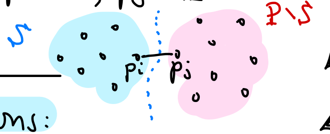


Algorithms for MST's:

- Based on greedy construction
- Add the lightest edge that causes no cycle

Kruskal's, Prim's, Borůvka's

Lemma: Given any cut $(S, P\setminus S)$ always safe to add lightest edge (p_i,p_j) $p_i\in S$, $p_j\in P\setminus S$

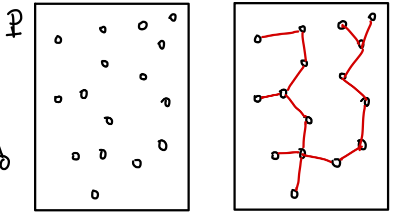


Applications:

- Clustering (Machine Learning)
- Approximation (TSP)
- Networking

Euclidean MST (EMST)

- The MST of P's Euclidean graph



Euclidean MSTs (II)

Prim's Algorithm:

- Given **point set P** + **start pt s_0**.
- **$S \subseteq P$**: Pts in spanning tree
  Init: $S = \{s_0\}$ End: $S = P$
- **$P \setminus S$**: Pts not yet in tree

```
while ($S \neq P$)
    - find closest ($p_i$, $p_j$)    $p_i \in S$    $p_j \in P \setminus S$
    - add $p_i$ to $S$
    - add ($p_i$, $p_j$) to tree
```

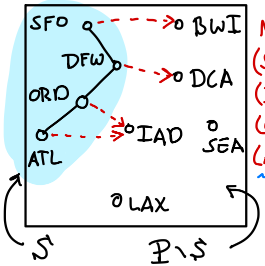
Finding next edge?

- **Brute force**: $O(n^2) \Rightarrow O(n^3)$ ☹
- **kd-tree**: To compute near neighbor
- **Priority queue**: To find best pair

Nearest-Neighbor Pairs:

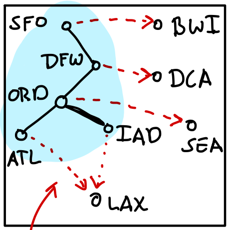
Given $p_i \in S$, let p_j be the closest point in $P \setminus S$

(p_i, p_j) is **nearest-neighbor pair**



NN:
(SFO,BWI)
(DFW,DCA)
(ORD,IAD)*
(ATL,IAD)

Add:
(ORD,IAD)



New NN Pairs:
(ATL,LAX) (IAD,LAX)
(ORD,SEA)

Given NN pair (p_i, p_j) we say p_i **depends** on p_j

Dependents list **dep(p_j)** is list of all pts p_i that depend on p_j

Point: p	dep(p)
BWI	{SFO}
DCA	{DFW}
SEA	{}
IAD	{ORD, ATL}
LAX	{}

How to do this?

- Lots of data structures!

List: Store edges of tree (e.g. {(SFO,DFW), (DFW,ORD), ... })

Set: Store points of S (e.g. {SFO, DFW, ORD, ATL})

Spatial Index: Stores pts of $P \setminus S$. Answers NN queries

Priority Queue: Stores the NN pairs ordered by squared dist. (E.g. (SFO,BWI) (DFW,DCA)...)

Hashmap of lists: Stores dependency lists, indexed by point

Basic Objects:

edgeList: list of edges in tree
inEMST: set representing S
kdTree, heap: ...
dependents: dep lists for all $P \setminus S$

Prim (Points P, Point start)

```
initialize (later)
add start to inEMST
nn ← kdTree.nearNeigh(start)
add start to dep[nn]
add new NN pair (start, nn)
while (kdTree ≠ ∅)
    edge ← heap.extractMin()
    if (edge.getSecond() ∉ inEMST)
        addEdge(edge) (later)
```

Euclidean MSTs (III)

Helpers:

- Initialize (Point start)
  - initialize all structures
- addEdge (Pair⟨Point⟩ edge)
  - add new edge to EMST
- addNN (Point pt, Point nn)
  - add new NN pair (pt, nn)

```
addEdge(Pair⟨Point⟩ edge)          (first, second)
    add edge to edgeList
    pt2 ← edge.getSecond()
    add pt2 to inEMST
    delete pt2 from kdTree
    dep2 ← get pt2 dep list from dependents
    for each (pt3 in dep2)
        nn3 ← kdTree.nearNeigh(pt3)
        if (nn3 == null) break
        addNN(pt3, nn3)
```

Q: Why check nn3 == null?
- On adding last pt to EMST the kd-tree is empty.

```
addNN(Point pt, Point nn)
    dist ← distanceSq(pt, nn)
    pair ← new Pair(pt, nn)
    insert pair in heap w. priority dist
    add pt to dep[nn]          Look up in hash map
```

```
initialize(Point start)
    clear: edgeList
           inEMST
           heap + kdTree
    for each (dep in dependents)
        clear dep
    for each(pt in P)
        if (pt ≠ start) insert pt in kdTree
```

That's it!

Is this efficient?
- Assuming NN queries in $\mathcal{O}(\log n)$ time

Total time = $\mathcal{O}(n \cdot \log n + m \cdot \log n)$

m = # of NN updates

↳ Much depends on m.
m depends on pt. distrib.