Introduction to Data Structures

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

Course Overview:

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

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?

Our approach:

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

Asymptotic Analysis:

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

Asymptotic: "Big-O"

- Ignore constants
- Focus on large n

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

$$T(n) = O(n^2)$$

Common:

$O(1)$: constant time ☺
[Hash map]

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

$O(n^p)$: (p = constant) Poly time
e.g. $O(\sqrt{n})$ [Geometric search]

Basic Data Structures I

- ADTs
- Lists, Stacks, Queues
- Sequential Allocation

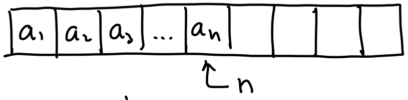
Linear List ADT:

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

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

Implementations:

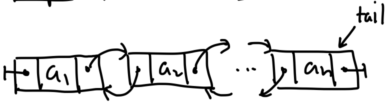
Sequential: Store items in an array



Linked allocation: linked list

Singly: head →

Doubly: head →



Performance varies with implementation

Abstract Data Type (ADT)

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

Doubling Reallocation:

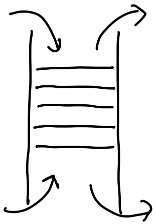
When array of size n overflows

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

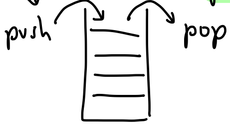
Dynamic Lists + Sequential Allocation:

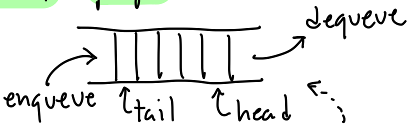
What to do when your array runs out of space?

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



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





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

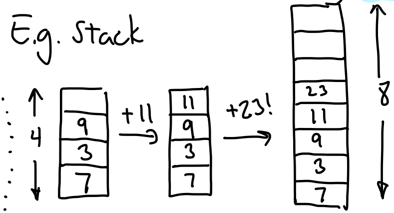
Basic Data Structures II

- Amortized analysis of dynamic stack

Dynamic (Sequential) Allocation

- When we overflow, double

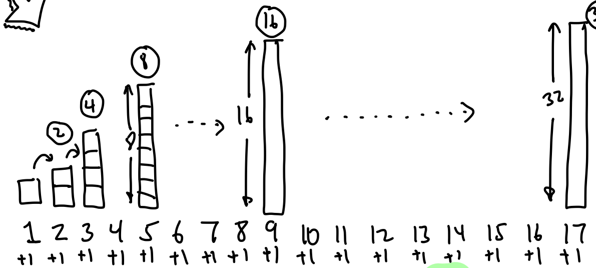
E.g. Stack



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.

[i.e. 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.

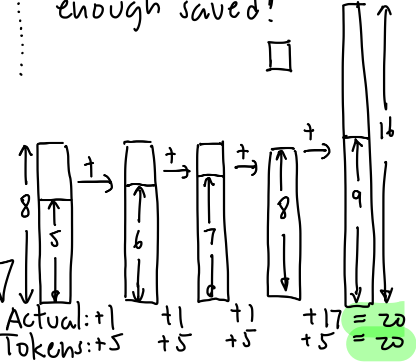
Proof:

- 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

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 $O(1)$ amortized cost per operation?

Leave as exercise

(Spoiler alert!)

Fixed increment → no

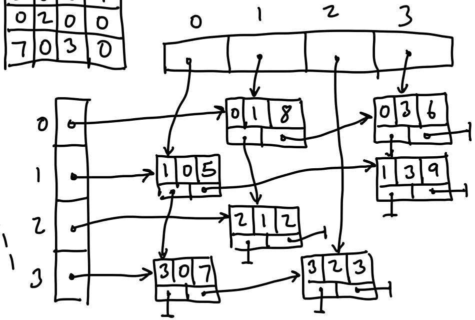
Fixed factor → yes

Squaring → ?? (depends on cost model)

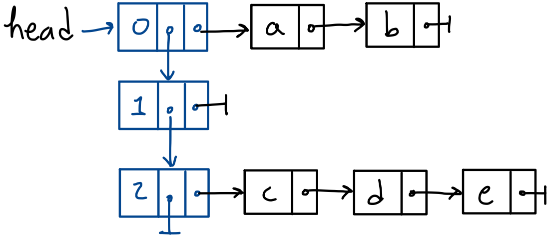
Dynamic Stack:

- Showed doubling ⇒ Amortized $O(1)$
- Other strategies?

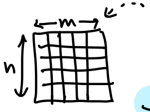
0	8	0	6
5	0	0	9
0	2	0	0
7	0	3	0



Multilists: Lists of lists

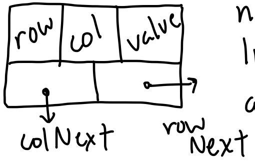


Sparse Matrices: An n×m matrix has n·m entries and takes (naively) $O(n \cdot m)$ space



Sparse matrix: Most entries are zero

Node:

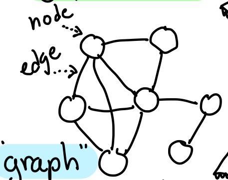


Idea: Store only nonzero entries linked by row and column

Trees: Basic Concepts and Definitions

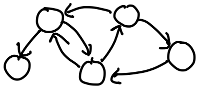
Graph: G = (V, E)
V = finite set of vertices (nodes)
E = set of edges (pairs of vertices)

Undirected



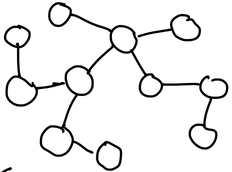
"graph"

Directed

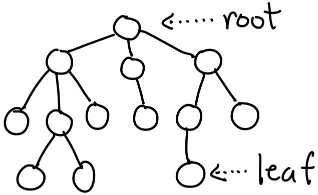


"digraph"

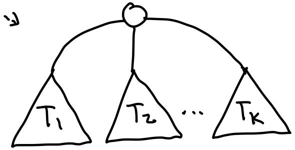
Tree (or "Free Tree")
- undirected
- connected
- acyclic graph



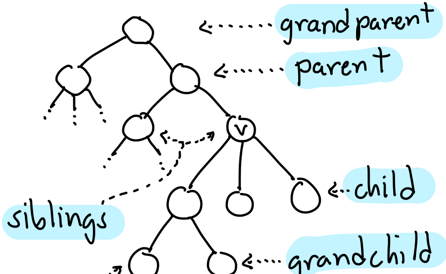
Rooted tree: A free tree with root node



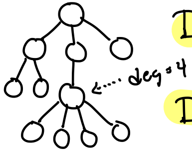
Formal definition:
Rooted tree: is either
- single node (root)
- set of one or more rooted trees ("subtrees") joined to a common root



"Family" Relations

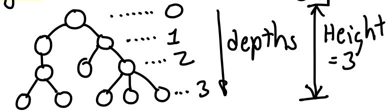


leaf: no children



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

Depth: path length from root
Height: (of tree) max depth

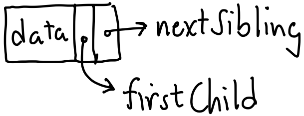


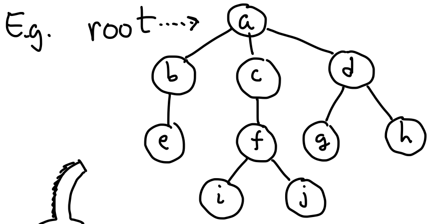
Trees Representation + Binary Trees (1)

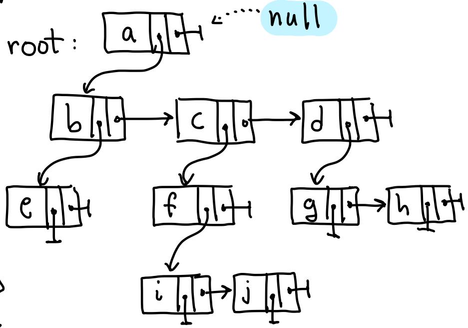
Representing rooted trees:

Each node stores a (linked) list of its children

Node structure:

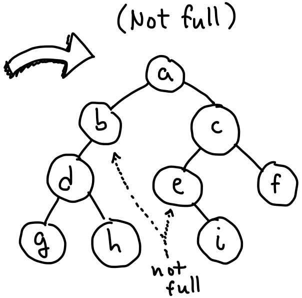


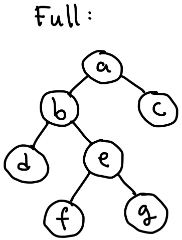




called the **Binary representation**

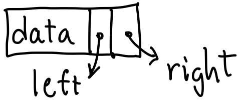
Binary tree: A rooted tree of degree 2, where each node has two children (possibly null) left + right

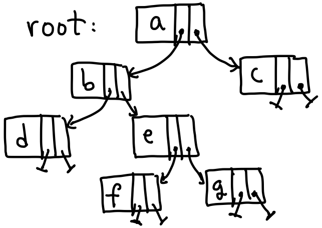




Full: Every non-leaf node has 2 children

Node structure:





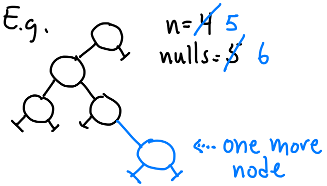
In Java:

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

generic data

Wasted space?

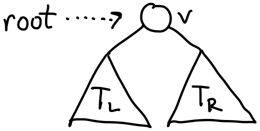
Theorem: A binary tree with n nodes has $n+1$ null links



Binary Trees: Traversals, Extension, and More

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

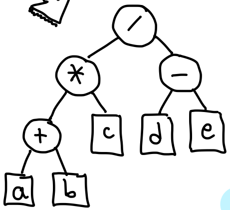
Binary Tree Traversals (can be generalized)



- 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
}
```



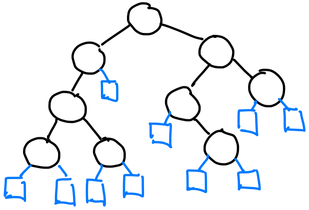
Preorder: / * + a b c - d e

Postorder: a b + c * d e - /

Inorder: a + b * c / d - e

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**

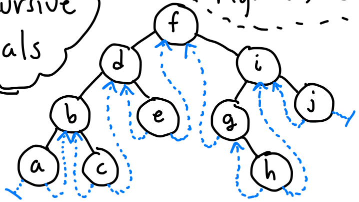
Another way to save space...

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

E.g. **Inorder Threads:**

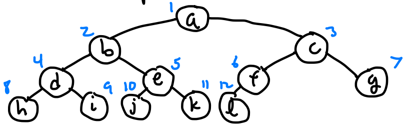
Null left → inorder predecessor

Null right → " successor



Challenge: Nonrecursive traversals

Complete Binary Tree: All levels full (except last)



a	b	c	d	e	f	g	h	i	j	k	l
1	2	3	4	5	6	7	8	9	10	11	12

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

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

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

Disjoint Set Union-Find I

Equivalence Relation:

Binary relation over set S such that $\forall a,b,c \in S$:

reflexive: $a \equiv a$

symmetric: $a \equiv b \Rightarrow b \equiv a$

transitive: $a \equiv b + b \equiv c \Rightarrow a \equiv c$

Any equivalence relation defines a **partition** over S.

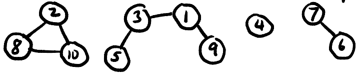
Examples:

- Given prime p, $a \equiv b \bmod p$

  E.g. $p=5$; **Partition**: $\{0,5,10,\dots\},\{1,6,11,\dots\}$ $\{2,7,12,\dots\}\dots$

- Given graph G, vertices u,v,

  $u \equiv v$ if in same connected component



Partition: $\{2,8,10\},\{1,3,5,9\},\{4\},\{6,7\}$

Union-Find:

Given set $S=\{1,2,\dots,n\}$ maintain a partition supporting ops:

Init: Each element in its own set $\{1\},\{2\},\dots,\{n\}$

Union(s,t): Merge two sets s + t and replace with their union

Find(x): Return the set containing x

Example:

Suppose: $\underset{s_1}{\{1,5\}}, \underset{s_2}{\{2,6,8\}}, \underset{s_3}{\{3,4,7\}}$

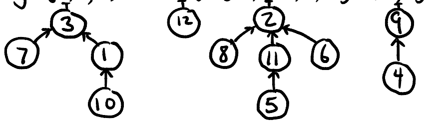
$\text{Union}(s_1,s_3) \rightarrow \{1,3,4,5,7\}, \{2,6,8\}$

$\text{Find}(5) \rightarrow s_1 \quad \text{Find}(8) \rightarrow s_2$

Inverted-Tree Approach:

- Store elements of each set in tree with links to **parent**
- Root node is **set identifier**

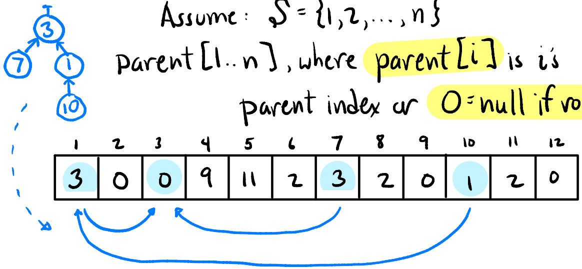
Eg. $\{1,3,7,10\}$ $\{12\}$ $\{2,5,6,8,11\}$ $\{4,9\}$



Array-Based Implementation:

Assume: $S=\{1,2,\dots,n\}$

parent[1..n], where **parent[i]** is i's parent index or **0 = null if root**



1	2	3	4	5	6	7	8	9	10	11	12
3	0	0	9	11	2	3	2	0	1	2	0

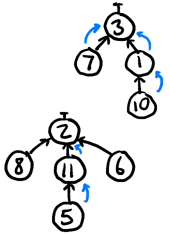
Set Identifiers are indices of root nodes

E.g. Find(7) = 3

Find(10) = 3

Find(5) = 2

Two items in same set iff Find(x) = Find(y)



A simple approach to finding is to trace the path to the root -

```
Set Simple-Find(Element x){
    while(parent[x] ≠ null)
        x ← parent[x]
    return x
```

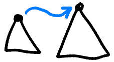
Set ≡ Element ≡ int

$1 \le x \le n$

Disjoint Set Union-Find II

How to Union?

- Just link one tree under the other



- How to maintain low heights?
- **Rank:** Based on height of tree. Link lower rank as child

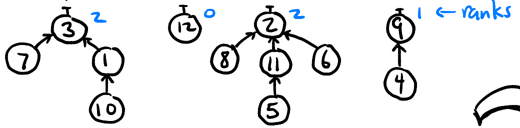
```
Set Union(Set s, Set t) {
    if (rank[s] > rank[t])
        swap s + t
    parent[s] ← t
    rank[t] ← max(rank[t], 1+ rank[s])
    return t
```

Recall: These are just array indices of roots

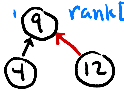
Init: All ranks ← 0

Example:

{1,3,7,10} {12} {2,5,6,8,11} {4,9}

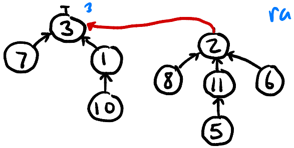


Union(9,12) [12 has lower rank]



$rank[9] = max(rank[9], 1+ rank[12]) = max(1,1) = 1$

Union(2,3) [Both have same rank]



$rank[3] = max(rank[3], 1+ rank[2]) = max(2,3) = 3$

Running Time?

Init: $O(n)$ - set a parents to null + ranks to 0

Union: $O(1)$ - constant time

Find: $O(\text{tree height})$

↳ What is worst case?

We'll show tree height = $O(\log n)$

⇒ Find takes $O(\log n)$ time

Lemma: Assuming rank-based merging a tree of height h has at least 2^h nodes.

Proof: By induction on num. of unions

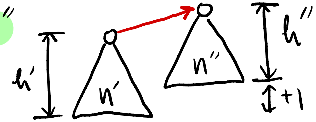
Basis: Single node. $h = 0$, $2^0 = 1$ nodes

Step: Consider the last of series of unions. Let T' + T'' be trees to merge: Heights: h' + h''

Sizes : n' + n''

By induction: $n' \geqslant 2^{h'}$ $n'' \geqslant 2^{h''}$

Cases: $h' = h''$



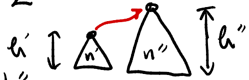
Final tree **height**: $h = h' + 1 = h'' + 1$

Final **size**:

$n = n' + n'' \geqslant 2^{h'} + 2^{h''} = 2^{h-1} + 2^{h-1}$

$= 2 \cdot 2^{h-1} = 2^h$ ✓

Case 2: $h' < h''$:



Final **height**: $h = h''$

Final **size**:

$n = n' + n'' \geqslant 2^{h'} + 2^{h''} \geqslant 2^{h''} = 2^h$ ✓

Case 3: $h' > h''$ (symmetrical) □

Disjoint Set Union-Find III

Simple Union-Find performs a sequence of m Unions + Finds on set of size n in $O(m \log m)$ time.

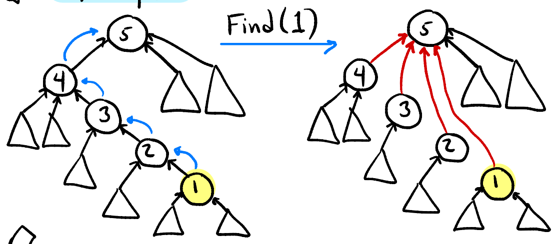
⇒ **Amortized time** (average per op) is $O(\log m)$

- Not bad - But can we do better?

Path Compression:

- Whenever we perform find, "short-cut" the links so they point directly to root
- This does not increase running by more than constant, but can speed up later finds

Example:



Does this little trick improve running times?

→ **Worst case** - No. Find may take $O(\log n)$ time

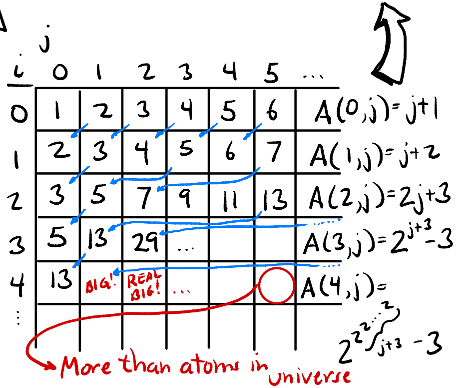
→ **Amortized** - Yes! Huge improvement! (But hard to prove)

Digression: Ackerman's Function (1926)

for $i, j \geq 0$

$$A(i,j) = \begin{cases} j+1 & \text{if } i = 0 \\ A(i-1,1) & \text{if } i>0, j=0 \\ A(i-1, A(i,j-1)) & \text{o.w.} \end{cases}$$

Looks innocent, but it's a monster!



i \ j	0	1	2	3	4	5	…	
0	1	2	3	4	5	6		$A(0,j) = j+1$
1	2	3	4	5	6	7		$A(1,j) = j+2$
2	3	5	7	9	11	13		$A(2,j) = 2j+3$
3	5	13	29	…				$A(3,j) = 2^{j+3} - 3$
4	13	BIG!	REAL BIG!	…				$A(4,j) = \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}_{j+3} - 3$
⋮								

More than atoms in universe

Obs: $\alpha(m,n) \leq 4$ for any imaginable values of m, n $(m \geq n)$

From super big to super small

Inverse of Ackerman:

$$\alpha(m,n) = \min \{ i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log m \}$$

Theorem: (Tarjan 1975) After init. any seq of m Union-Finds (with path compression) takes total time $O(m \cdot \alpha(m,n))$.

⇒ **Amortized time** $= O(\alpha(m,n))$

[For all practical purposes, this is constant time.]

Priority Queues + Heaps I

Priority Queue:

- Stores key-value pairs
- Key ≡ priority
- Ops: insert(x,v) - insert value v with key x
  extract-min - remove/return pair with min key value

Naive Solution:

- Store items in linear list
- Order?
  - Insert order - fast insert / slow extract
  - Priority order - fast extract / slow insert

Heap: Tree-based structure

(min) heap order: for all nodes, parent's key ≤ node's key

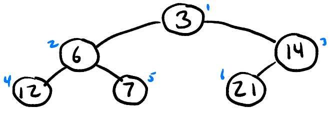
[Reverse: max-heap order]

Many variants: Binary, leftist, binomial, Fibonacci, pairing, quake, skew ... heaps

Binary Heap:

- simple, elegant, efficient
- old (1964)
- basic: insert/extract - $O(\log n)$
  build - $O(n)$

"Pointerless" tree



A:	X	3	6	14	12	7	21	
	0	1	2	3	4	5	6	..

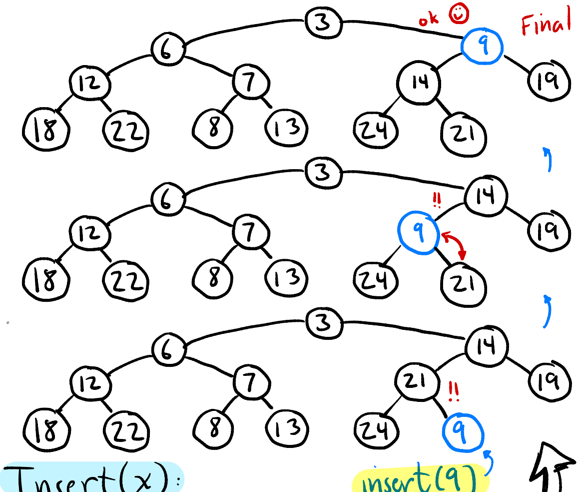
left(i): if($2\cdot i \le n$) $2\cdot i$ else null

right(i): if($2\cdot i+1 \le n$) $2\cdot i+1$ else null

par(i): if ($i \ge 2$) $\lfloor i/2 \rfloor$ else null

Insert(x):

- Append x to end of array
- Sift x up until its parent's key is smaller (or reaching root)



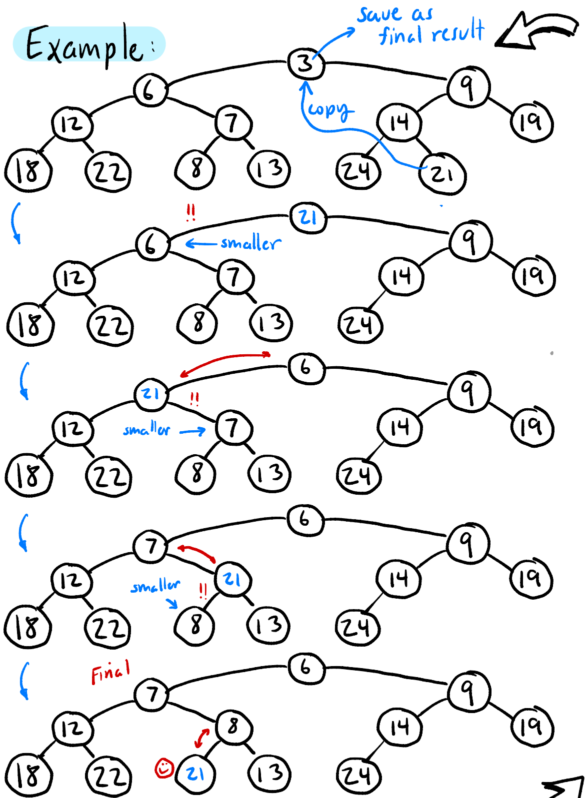
```
void insert(Key x)            (ignore value)
    n++; i ← sift-up(n,x)
    A[i] ← x
int sift-up(int i, Key x)
    while (i>1 && x<A[par(i)])
        A[i] ← A[par(i)]
        i ← par(i)
    return i
```

Priority Queues + Heaps II

Binary Heap - Extract Min

- Min key at root → save it
- Copy A[n] to root (A[1]) + decrement n
- Sift the root key down
  - find smaller of two children
  - if larger, swap with this child
- Return saved root key

Example:



```
Key extract-min()
    if (n == 0) Error - Empty heap
    result ← A[1]
    z ← A[n--]      // get replacement
    i ← sift-down(i, z)
    A[i] ← z
    return result
```

```
int sift-down(int i, Key z)
    while (left(i) ≠ null)
        u ← left(i);  v ← right(i)
        if (v ≤ n && A[v] < A[u])
            u ← v      // A[u] is smaller child
        if (A[u] < z)
            A[i] ← A[u]; i ← u
        else break
    return i
```

Leftist Heaps: Meldable heaps

- Can merge two heaps into single heap
- E.g. One processor breaks. Awaiting jobs must be merged with another processor.

Analysis: Both insert + extract-min take time proportional to tree height. Tree is complete ⇒ $O(\log n)$ time

Leftist Property: Null path length

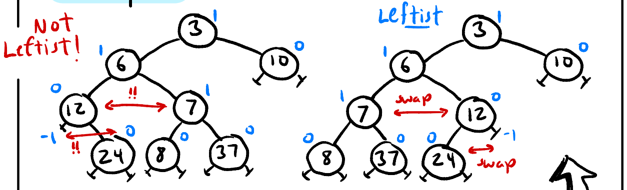
npl(v) = length of shortest path to null

$$\text{npl}(v) = \begin{cases} -1 & \text{if } v = \text{null} \\ 1 + \min\left(\begin{array}{l}\text{npl}(v.\text{left})\\ \text{npl}(v.\text{right})\end{array}\right) & \text{o.w.} \end{cases}$$

Def: Leftist Heap is binary tree where:

- Keys are heap ordered
- ∀ nodes v, npl(v.left) ≥ npl(v.right)

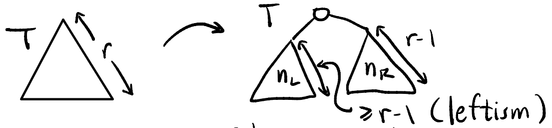
Examples:



Priority Queues + Heaps III

Lemma: A leftist tree with $r \geq 1$ nodes along its rightmost path has $n \geq 2^r - 1$ nodes

Proof: (Sketch - see latex notes)



By induction: $n_L \geq 2^{r-1}-1$, $n_R \geq 2^{r-1}-1$

$n = 1 + n_L + n_R \geq (2 \cdot 2^{r-1} - 1) + 1 = 2^r - 1$ □

Class structure:

```
LeftistHeap<Key>{
    private class LHNode{
        Key x
        LHNode left, right
        int npl
    }
    private LHNode root
    public LeftistHeap() {root ← null}
      "     void insert(Key x)
      "     Key extractMin()
      "     void mergeWith(LeftistHeap H2)
    ... (other private/protected utilities)
```

inner class - used only by LeftistHeap

references root node

constructor

public functions

```
public mergeWith(LeftistHeap H2){
    root ← merge(this.root, H2.root)
    H2.root ← null
}
```

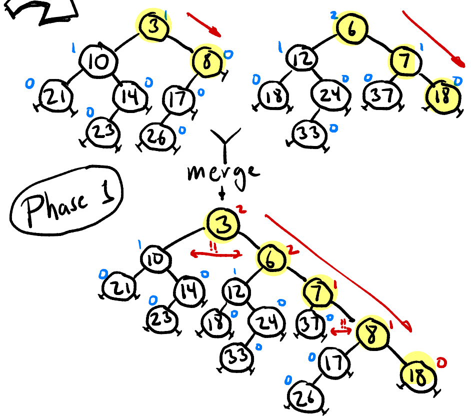
helper function

merger destroys H2

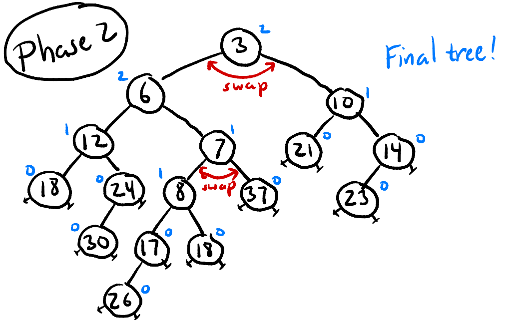
Merge helper: 2 phases

① Merge right paths by order of keys + update npl's

② Check leftist property + swap



```
LHNode merge(LHNode u, LHNode v){
    if (u == null) return v
    if (v == null) return u
    if (u.key > v.key)      // swap so u is smaller
        swap u ↔ v
    if (u.left == null) u.left ← v
    else
        u.right ← merge(u.right, v)
        if (u.left.npl < u.right.npl)
            swap u.left ↔ u.right
        u.npl ← u.right.npl + 1
    return u
```



Analysis: Time ∼ Rightmost path $= O(\log n)$

Insert + Extract-min? Exercises

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)

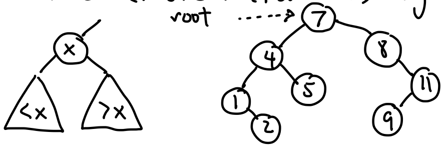
- returns a reference to associated value v, or **null** if not there.

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

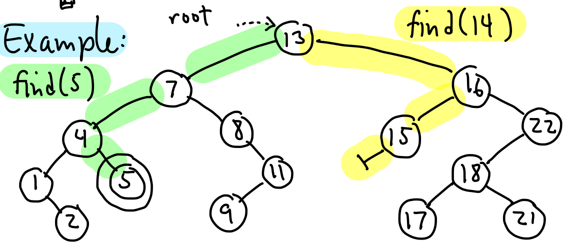


Find: How to find a key in the tree?

- 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!

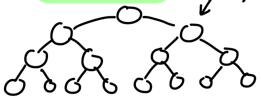
```
Value find(Key x, BSTNode p) {
    if (p == null) return null
    else if (x < p.key)
        return find(x, p.left)
    else if (x > p.key)
        return find(x, p.right)
    else return p.value
}
```

Example: find(5) find(14)

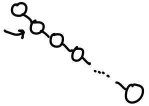


Efficiency: Depends on tree's height

Balanced: O(log n)



Unbalanced: O(n)

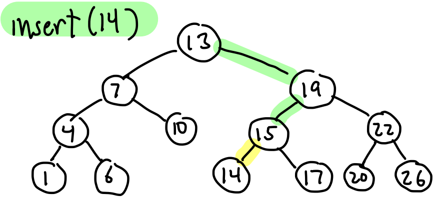


Binary Search Trees II

- insertion
- deletion

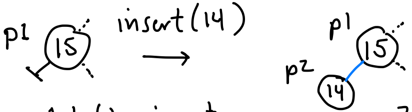
Insert (Key x, Value v)

- 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!

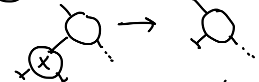
Delete (Key x)

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

How?

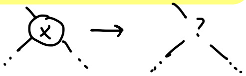
3 cases:

① ⓧ is a leaf



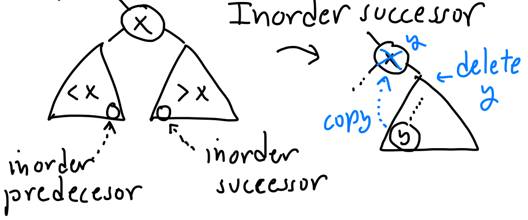
② ⓧ has single child

3. ⓧ has two children



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

Replacement Node?

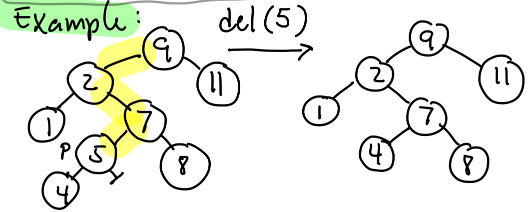


Binary Search Trees III

- 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
}
```

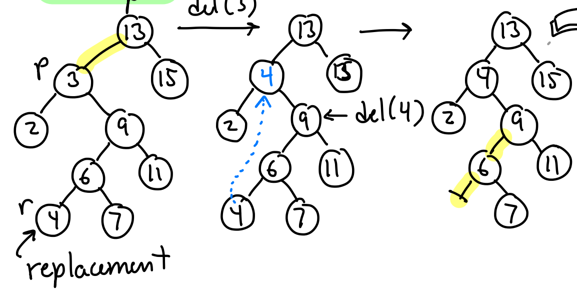
Example:



Find Replacement Node

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

Example:



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

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).

Analysis:

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

Java implementation (see notes for details)

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

        .... constructor, toString...
    }

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

    private Node root;

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

Inner class for node (protected)

Local helpers (private or protected)

Data (private)

Public members (invoke helpers)

AVL Trees I

- Basic defs
- Height props
- Rotations

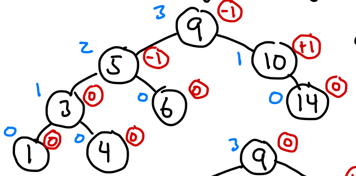
AVL Height Balance

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

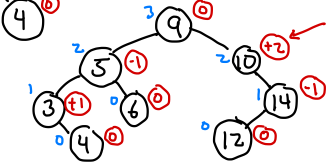
AVL tree: A binary search tree that satisfies this condition

Balance factor:

$$bal(v) = hgt(v.right) - hgt(v.left)$$



← This is an AVL tree

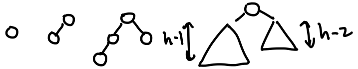


← Not an AVL tree

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)

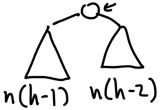
$h = 0$: $n(h) = 1 = F_3 - 1$

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

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

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

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



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

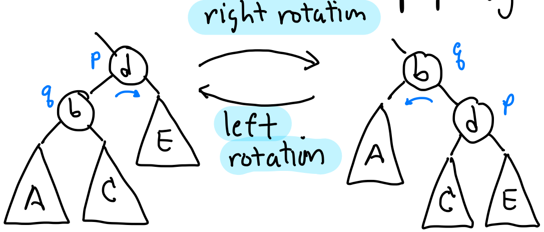
Proof: Fact: $F_h \approx \varphi^h / \sqrt{5}$ where $\varphi = (1+\sqrt{5})/2$ "Golden ratio"

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

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

$= O(\log n)$ □

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

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)
```

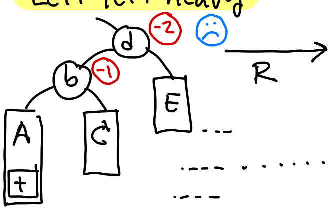
Find: Same as B.S.T.

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

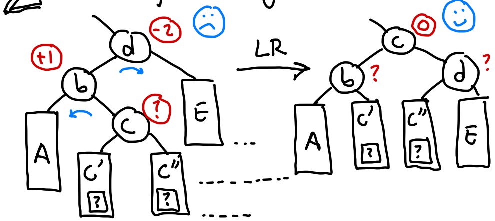
How to rebalance?

Bal = -2

Left-left heavy



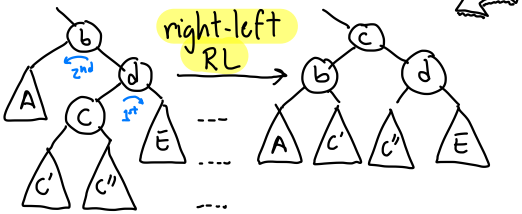
Left-right heavy:

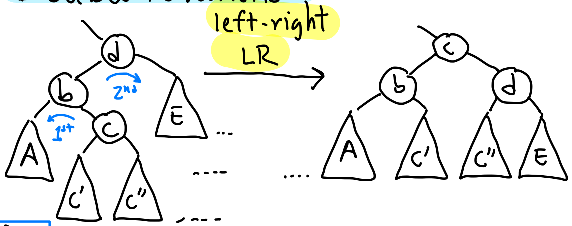


```
AVLNode insert(Key x, Value v, AVLNode p) {
    if (p == null) p = new AVLNode(x, v)
    else if (x < p.key)
        p.left = insert(x, v, p.left)
    else if (x > p.key)
        p.right = insert(x, v, p.right)
    else throw -Error-Duplicate!
    return rebalance(p)
```

```
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
```

Double rotations:





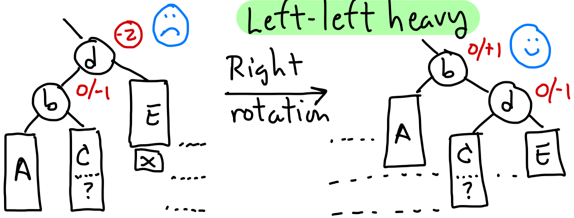
```
BSTNode rotateLeftRight(BSTNode p)
    p.left = rotateLeft(p.left)
    return rotateRight(p)
```

AVL Trees III

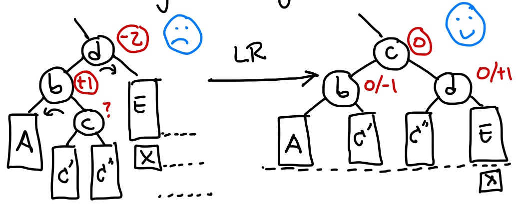
- Deletion
- Examples

Cases: Balance factor -2

Left-left heavy

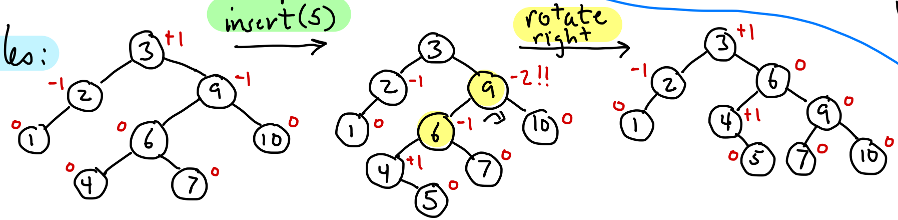


Left-right heavy



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

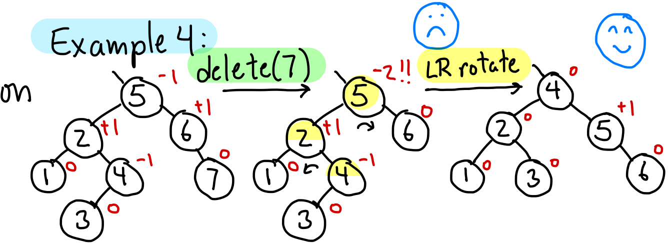
Examples:



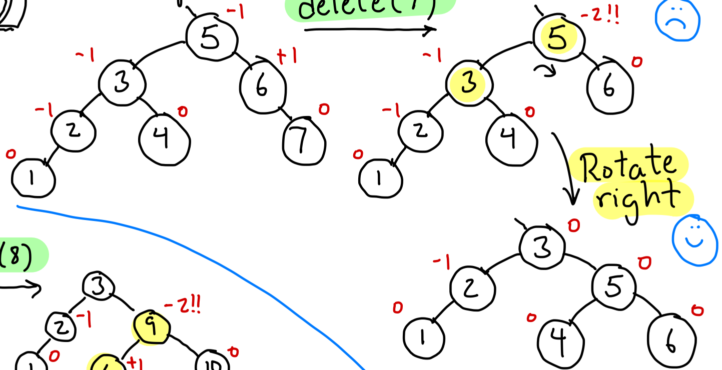
Deletion: Basic plan

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

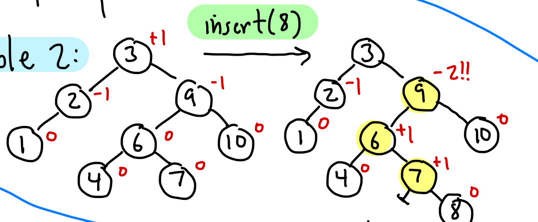
Example 4:

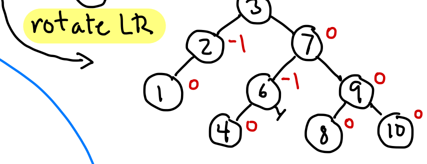


Example 3:

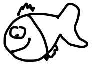


Example 2:





2-3 Trees I



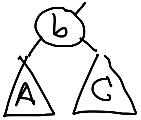
Recap:

AVL: Height balanced Binary

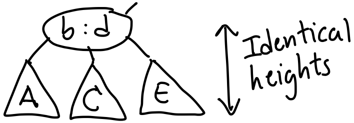
2-3 tree: Height exact Variable width

Node types:

2-Node
1 key
2 children



3-Node
2 keys
3 children

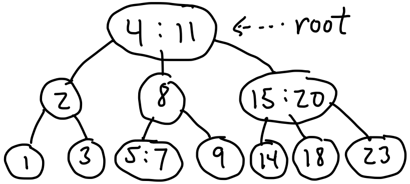


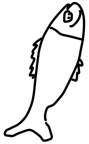
Def: A **2-3 tree** of height h is either:

- 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



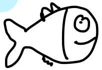


Thm: A 2-3 tree of n nodes has height $O(\log n)$

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

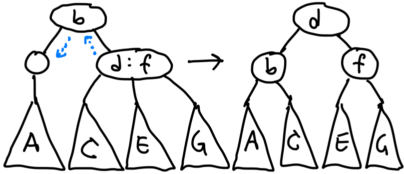
How to maintain balance?

- Split
- Merge
- Adoption (Key rotation)

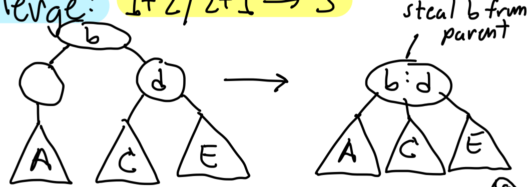


Adoption (Key-Rotation)

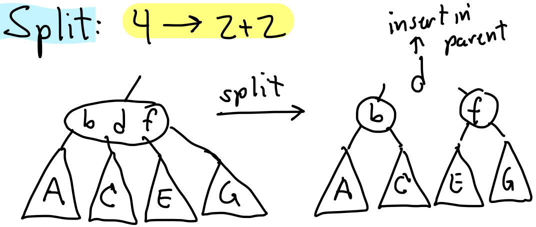
$1+3 = 2+2$



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

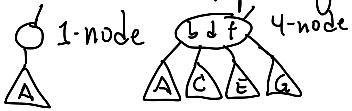


Split: $4 \rightarrow 2+2$



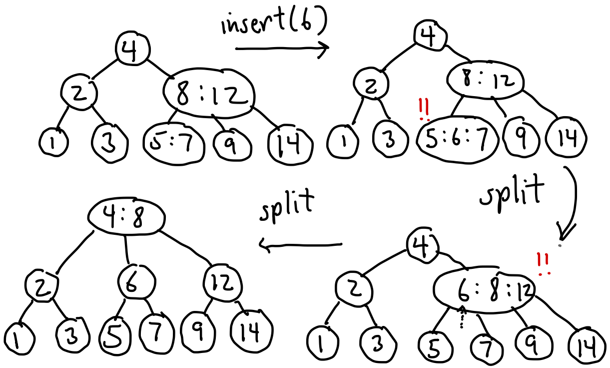
Conceptual tool:

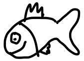
We'll allow 1-nodes + 4-nodes temporary



2-3 Trees II

Insertion example:





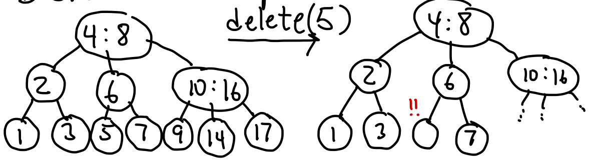
Dictionary operations:

- **Find** - straightforward
- **Insert** - find leaf node where key "belongs" + add it (may split)
- **Delete** - find / replacement / merge or adopt

Implementation?

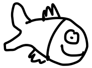
```
class TwoThreeNode {
    int nChildren
  TwoThreeNode children[3]
    Key key[2]
}
```

Delete Example:

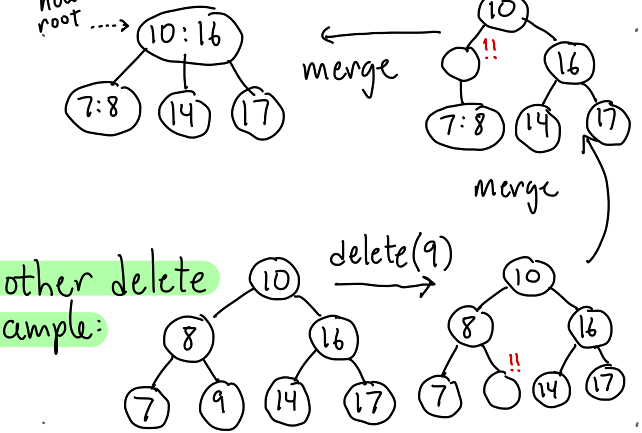


Deletion remedy:

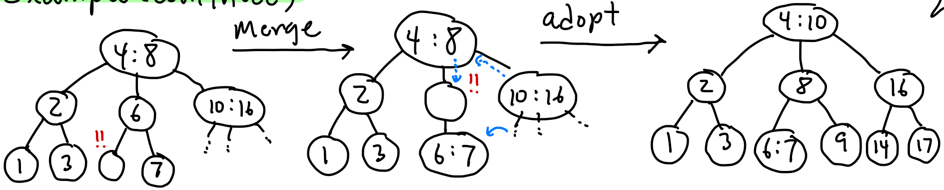
- Have a 3-node neighboring sibling → **adopt**
- o.w.: **Merge** with either siblings + steal key from parent



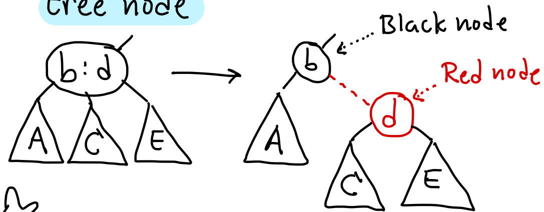
Another delete example:

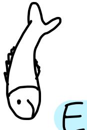


Example (continued)

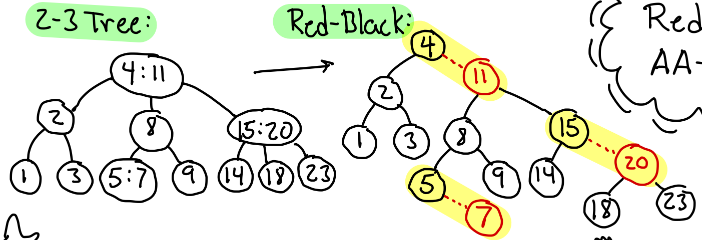


Encoding 3-node as binary tree node



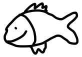


Example:



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



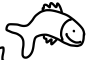
Some history:

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 $O(\log n)$

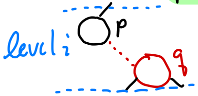
Proof: It's at most twice that of a 2-3 tree.

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

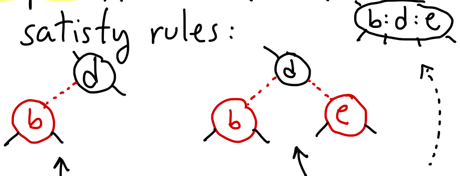
What we need are stricter rules!

AA-tree: Arne Anderson 1993

New rule:

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

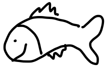
Nope! Alternatives that satisfy rules:



A "**left-skewed**" encoding

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

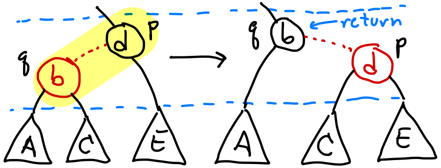
Red-Black + AA Trees II



Restructuring Ops:

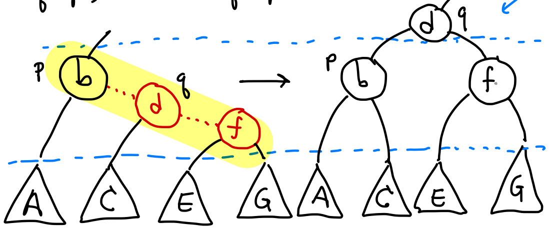
Skew: Restore right skew

→ 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 ?

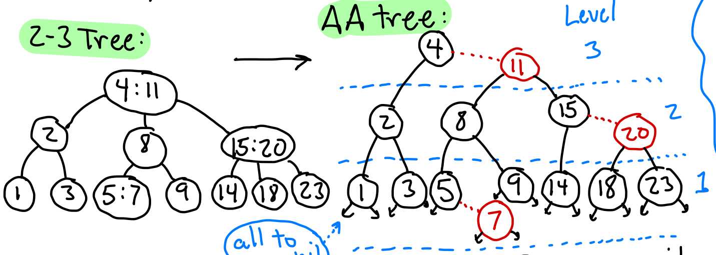
How to test?

p.level == p.right.level == p.right.right.level

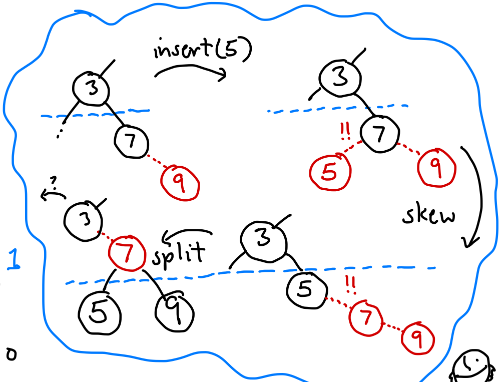
$\underbrace{\text{p.level == p.right.level}}$ not needed (levels are monotone)

Example:

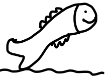
2-3 Tree: → **AA tree:**



all to nil



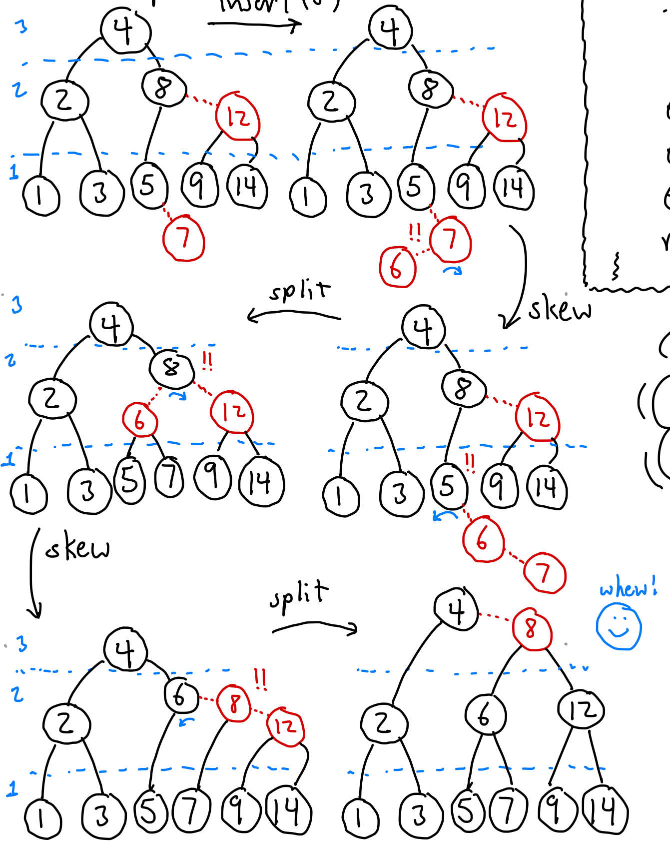
AA Insertion:
- Find the leaf (as usual)
- Create new red node
- Back out applying skew + split

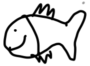


```
AANode skew(AANode p) {
    if (p == nil) return p
    if (p.left.level == p.level) {          right rotate p
        AANode q = p.left                 ←
        p.left = q.right ; q.right = p
        return q  ← new subtree root
    } else return p  ← everything's fine
}
```

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

Example:





```
AANode insert(Key x, Value v, AANode p) {
    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))
}
```

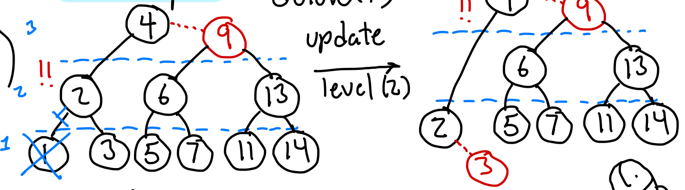
leaf level

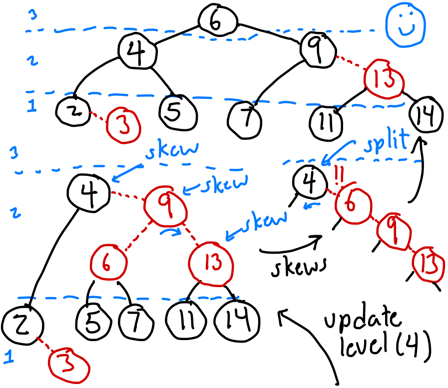
left, right

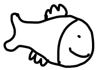
Red-Black and AA Trees III

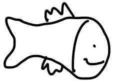
Example:

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)**

Treaps I

Randomized Data Structures

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

History:

1989: Seidel + Aragon
[Explosion of randomized algorithms]

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

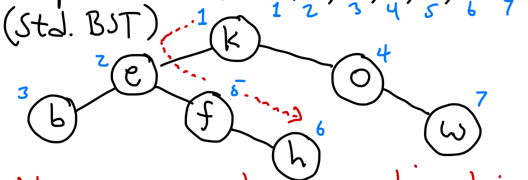
Intuition:

- Random insertion into BSTs ⇒ $O(\log n)$ 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

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

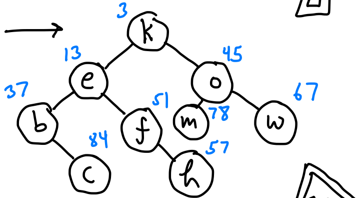
Treap: Each node stores a key + a random priority.
Keys are in inorder.
Priorities are in heap order

? Is it always possible to do both?

Yes: Just consider the corresponding BST

Example:

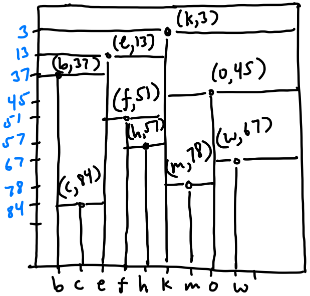
Key	Priority
b	37
c	84
e	13
f	51
h	57
k	3
m	78
o	45
w	67



Geometric Interpretation:

key → x
priority → y

McCreight's Priority Search Tree

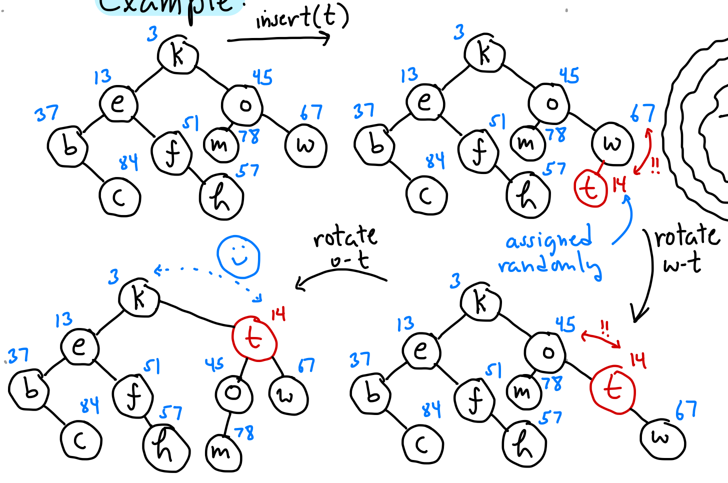


Treaps II

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 $+\infty$. 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)

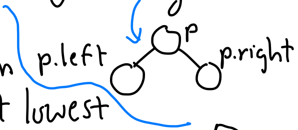
Proof: Follows directly from BST analysis

Implementation: (See pdf notes)

Node: Stores priority + usual...

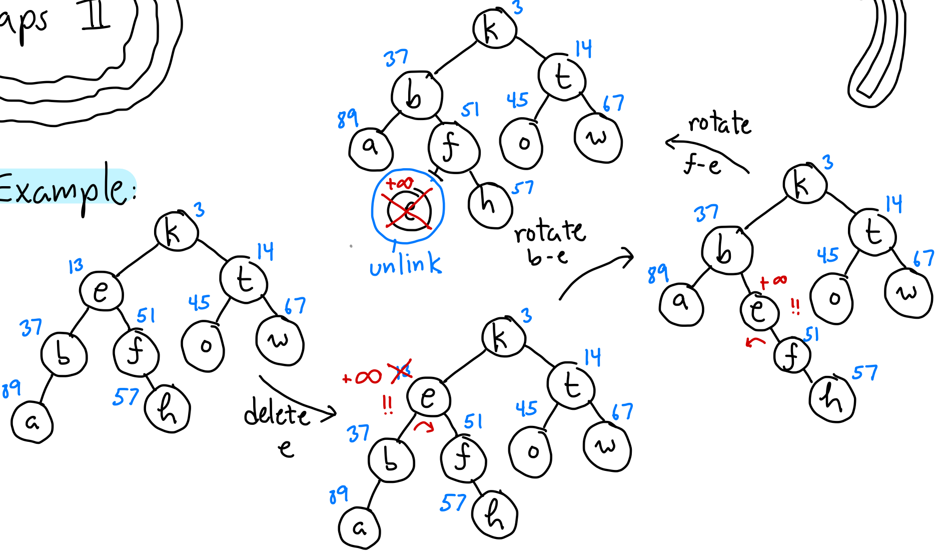
Helpers:

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



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

Example:



Skip Lists I

Ideal Skip List:

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

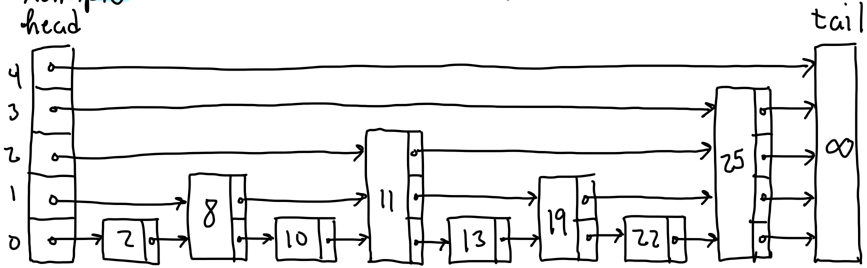
Sorted linked lists:

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

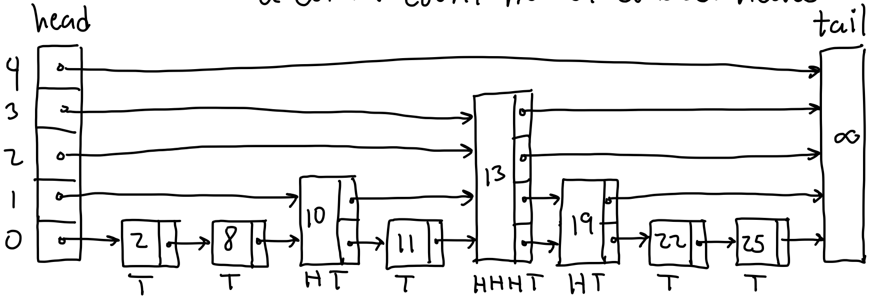
Idea: Add extra links to skip

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

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 $\le 1/n^{c-1}$.

→ 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 $\le n/2^{\ell}$
[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:

$$\le n/2^{\ell} = n/2^{(c\cdot\lg n)} = n/(2^{\lg n})^{c} = n/n^{c} = 1/n^{c-1}$$

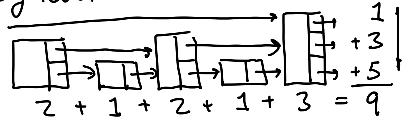
□

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

Skip Lists II

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 $\ge i$ is $1/2^{i}$
- Expected no. of nodes that contrib. to level $i = n/2^{i}$

⇒ $E(n_i) = n/2^{i}$

Total space (expected) is:

$$E\left(\sum_{i=0}^{\infty} n_i\right) = \sum_{i=0}^{\infty} E(n_i) = \sum_{i=0}^{\infty} n/2^{i} = n\sum_{i=0}^{\infty} 1/2^{i} = 2n$$

□

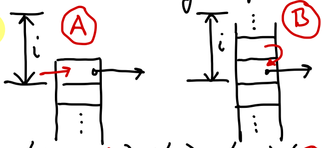
Thm: Expected search time is O(log n)

Proof:
- We have seen no. levels is O(log n)
- Will 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:



$$E(i) = 1 + (\text{Prob}(\text{Ⓐ}))E(i) + (\text{Prob}(\text{Ⓑ}))E(i-1)$$

current node ↑ same level ↑ from prior level

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

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

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

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

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

⇒ We visit 2 nodes per level on average.

□

Skip Lists III

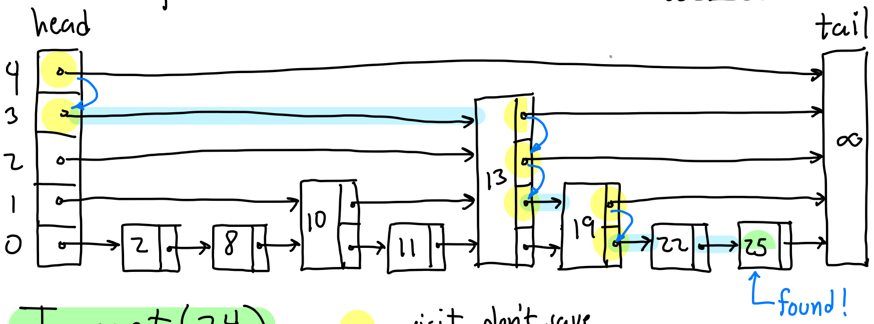
Delete:

- 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

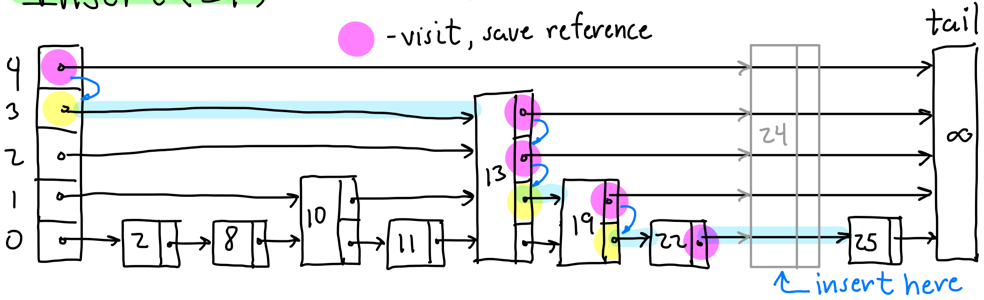
Example: find(25)



Insert(24)

🟡 - visit, don't save

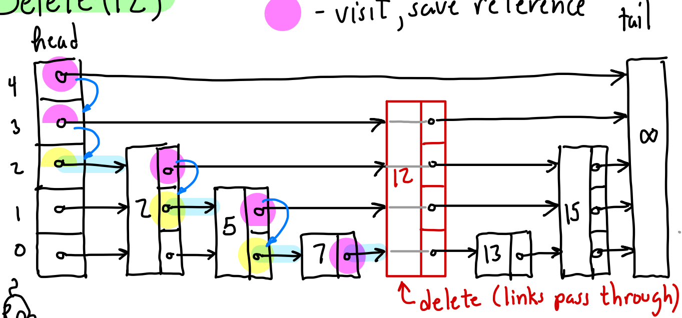
🟣 - visit, save reference



Delete(12)

🟡 - visit, don't save

🟣 - visit, save reference



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

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

Other/Better Criteria?

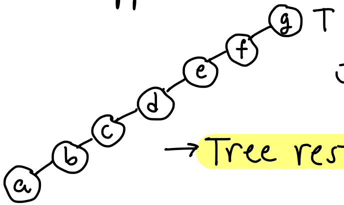
Expected case: Some keys more popular than others

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

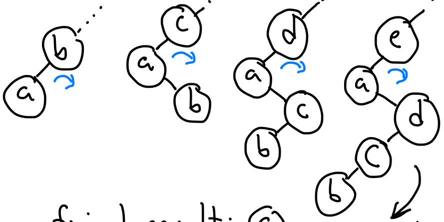


find("a")

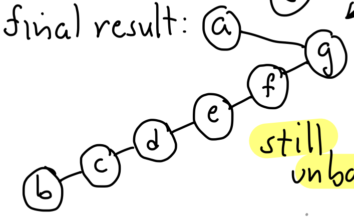
ugh!

→ Tree restructures itself

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

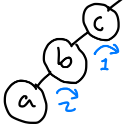


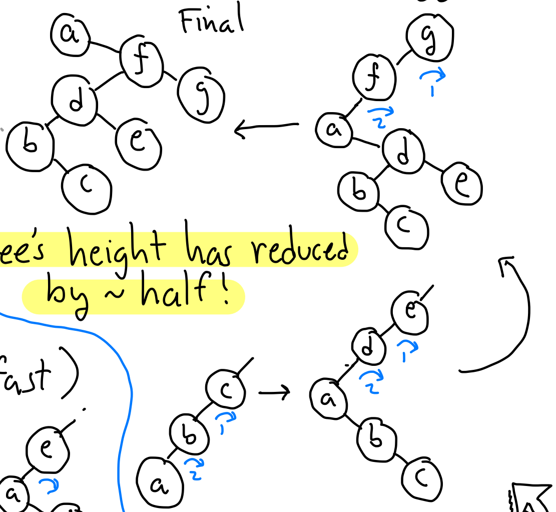
.... final result:



still unbalanced!

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





Tree's height has reduced by ~ half!

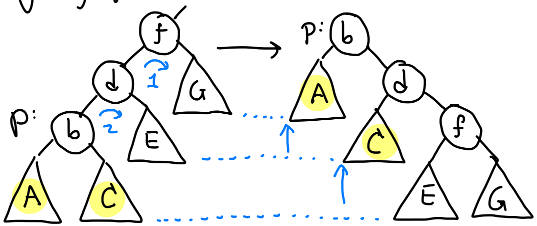
Lesson: Different combinations of rotations can:
- bring given node to root
- significantly change (improve) tree structure.

Splay Trees II

Splay(Key x):

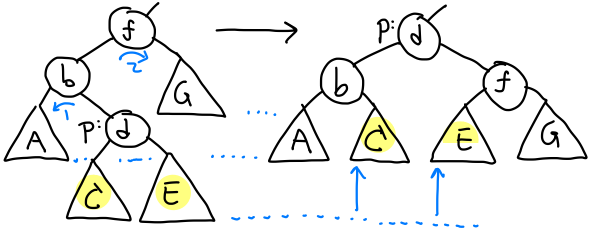
```
Node p ← find x by standard BST search
while (p ≠ root) {
    if (p == child of root) zig(p)
    else /* p has grand parent */
        if (p is LL or RR grand child) zigzig(p)
        else /* p is LR or RL gr. child */ zigzag(p)
```

ZigZig(p): [LL case]



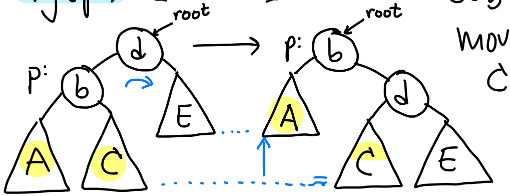
Subtrees A,C move up↑

ZIG ZAG(p): [LR case]



Subtrees C,E of p move up↑

Zig(p): [L case]



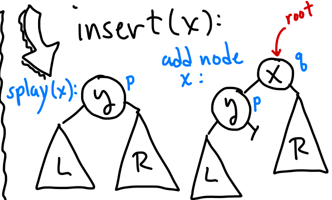
Subtree A moves up↑
C unchanged

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
```

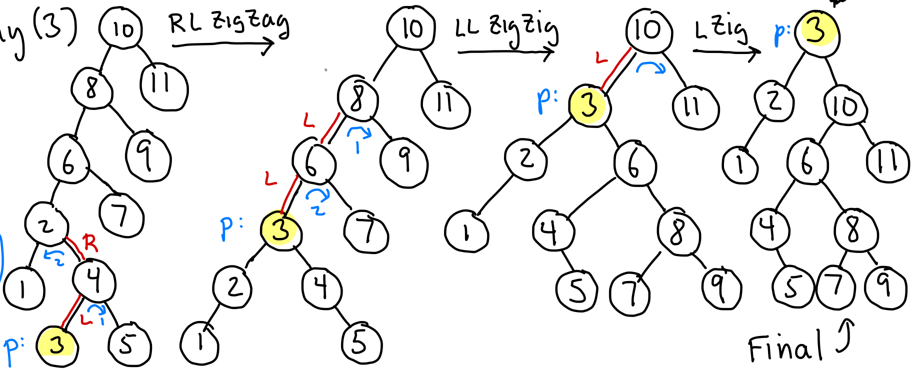
find(x):

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



Example:

splay(3)



SPLAY TREES III

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 $\left(\sum_{i=1}^{n} p_i = 1\right)$
- **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 $x_1 \ldots x_n$, where x_i is accessed q_i times. Let $p_i = q_i/m$. Then total time is $O(m \sum_i p_i \lg 1/p_i)$

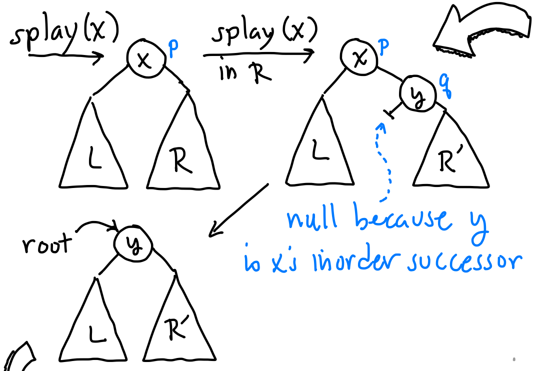
Dynamic Finger Theorem:

Keys: $x_1 < \ldots < x_n$. We perform accesses $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$. Let $\Delta_j = i_j - i_{j-1}$: distance between consecutive items

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

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 == null]
q.left = p.left
root = q
```



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

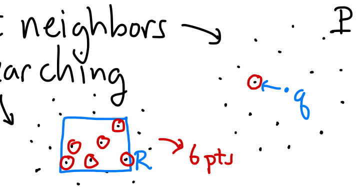
Quadtrees & kd-Trees I

So far: 1-dimensional keys

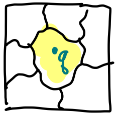
- Multi-dimensional data
- Applications:
  - Spatial databases + maps
  - Robotics + Auton. Systems
  - Vision/Graphics/Games
  - Machine Learning
- ...

Geometric Search:

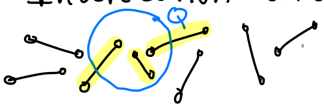
- Nearest neighbors
- Range searching



- Point Location



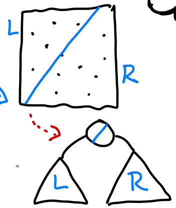
- Intersection Search



Multi-Dim vs. 1-dim Search?

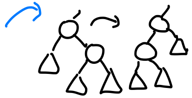
Similarities:

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



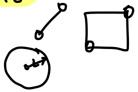
Differences:

- 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

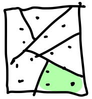


Point: A d-vector in $\mathbb{R}^d$

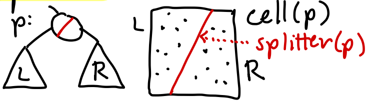
$p = (p_1, \dots, p_d) \quad p_i \in \mathbb{R}$

```
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 ...
}
```

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

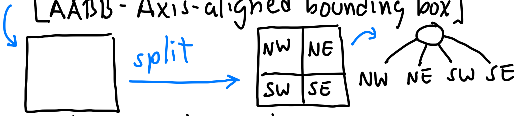


- External nodes store pts.

Quadtrees & kd-Trees II

Quadtrees: (abstractly)

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

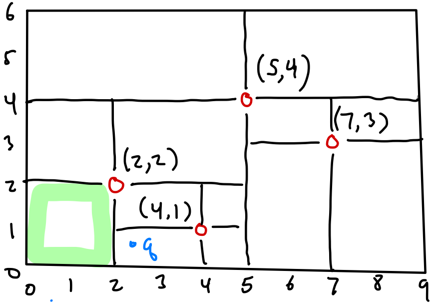


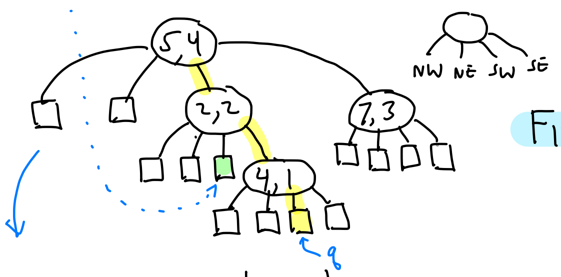
- **Splitter**: Subdivides cell into four (genlly 2^d) subcells

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

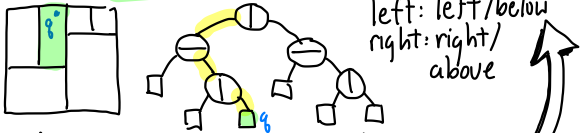
- Numerous variants! PR, PMR, QR, QX, ... see Samet's book
- Popular in 2-d apps (in 3-d, octtrees)
- Don't scale to high dim
  - out degree = 2^d
- What to do for higher dims?

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

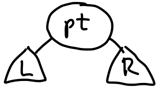


Quadtrees & kd-Trees III

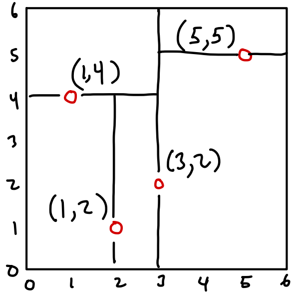
Kd-Tree Node:

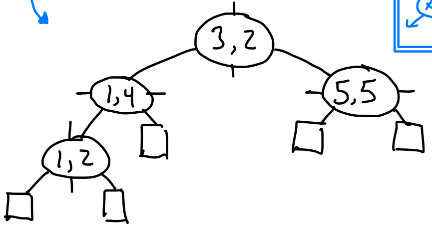
```
class KDNode {
    Point pt      // splitting point
    int cutDim    // cutting coordinate
    KDNode left   // low side
    KDNode right  // high side
}
```

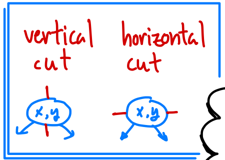
Example:



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

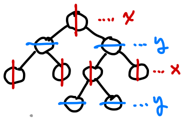




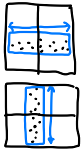


How do we choose cutting dim?

- **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.



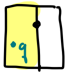
Find point q in subtree rooted at p with cutDim cd:

- 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)
}
```

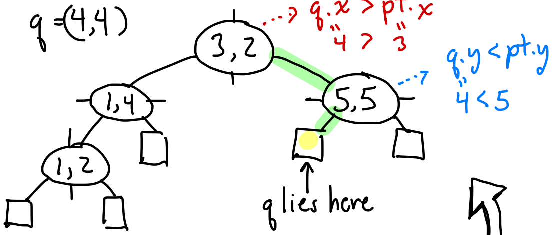


Theorem: If pts are inserted in random order, expected height is $O(\log n)$

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

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

q = (4,4)



Quadtrees & kd-Trees IV

Kd-Tree Insertion:

(Similar to std. BSTs)

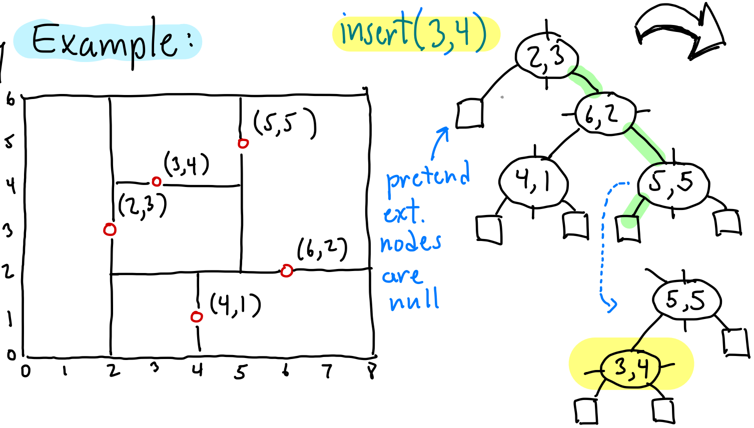
- Descend tree until
  - → find pt → Error - duplicate
  - → falling out ← (Although we draw extended trees, let's assume standard trees)
    - → create new node
    - → set cutting dim

```
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:



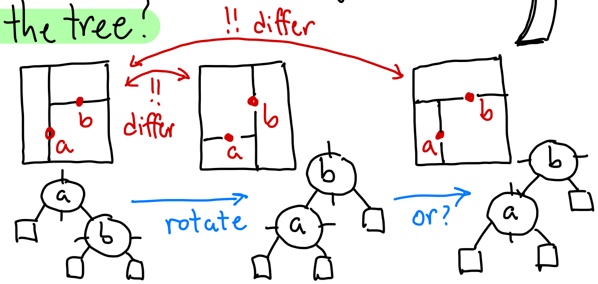
Analysis:

Run time: O(h)

Tree height

Can we balance the tree?

- Rotation does not make sense !!



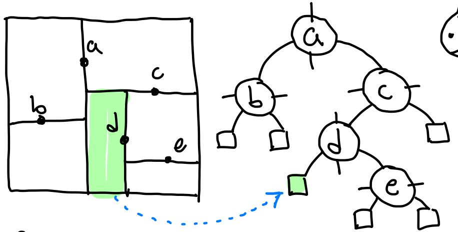
Rebalance by Rebuilding:

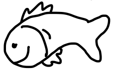
- Rebuild subtrees as with scapegoat trees
- O(log n) amortized
- Find: O(log n) guaranteed.

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.

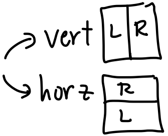




Kd-Tree Queries I

Kd-Trees:

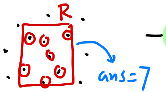
- Partition trees
- Orthogonal split



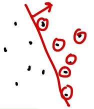
- Alternate cutting dimension x, y, x, y, …
- Cells are axis-aligned rectangles (AABB)

Queries?

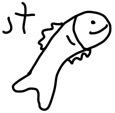
- Orthogonal range queries
  - Given query rect. (AABB) count/report pts in this rect.



- 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

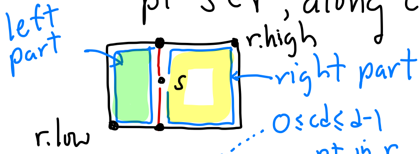


This Lecture: $O(\sqrt{n})$ time alg for orthog. range counting queries in $\mathbb{R}^2$

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

Rectangle methods for kd-cells:

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

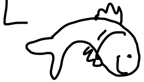


r.leftPart(cd, s)

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

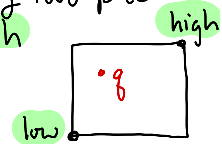
r.rightPart(cd, s)

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



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

- Defined by two pts: low, high



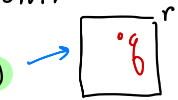
- Contains pt $q \in \mathbb{R}^d$ iff

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

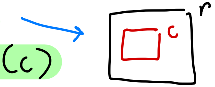
Useful methods:

Let r, c - Rectangle
q - Point

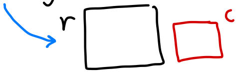
r.contains(q)



r.contains(c)

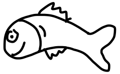


r.isDisjointFrom(c)



Kd-Tree Queries II

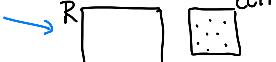
Orthog. Range Query



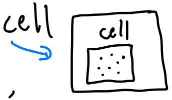
- Assume: Each node p stores:
  - **p.pt** : splitting point
  - **p.cutDim** : cutting dim
  - **p.size** : no. of pts in p's subtree
- Tree stores ptr. to **root** and **bounding box** for all pts.
- Recursive helper stores current **node p** + **p's cell**.

Cases:

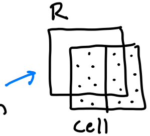
- p == null → fell out of tree → 0
- Query rect is **disjoint** from p's cell
  → return 0
  → no point of p contributes to answer



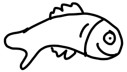
- Query rect **contains** p's cell
  → return p.size
  → every point of p's subtree contributes to answer.

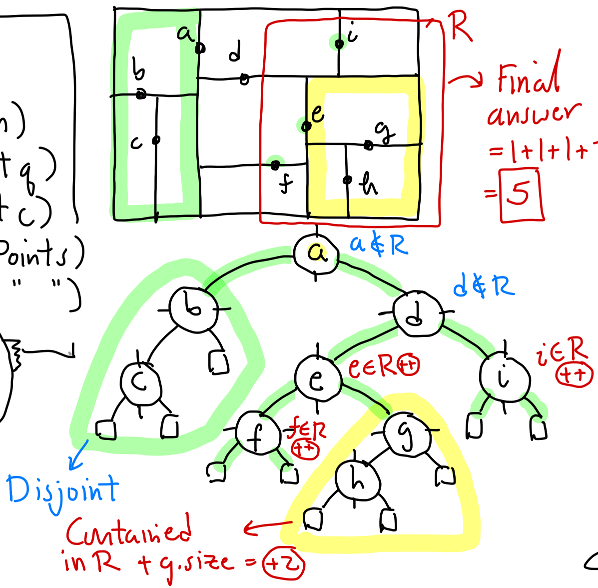


- Otherwise: Rect. + cell **overlap** – Recurse on both children



```
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("  "    "  ")
}
```

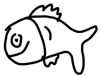




```
int rangeCount(Rect R, KDNode p, Rect cell)
  if (p == null) return 0 // fell out of tree
  else if (R.isDijointFrom(cell)) return 0 // no overlap
  else if (R.contains(cell)) return p.size // take all
  else { int ct = 0
     if (R.contains(p.pt)) ct++ // p's pt in range
     ct += rangeCount(R,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

Kd-Tree Queries III

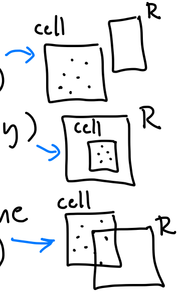
Theorem: Given a balanced kd-tree storing n pts in $\mathbb{R}^2$ (using alternating cut dim), orthog. range queries can be answered in $O(\sqrt{n})$ time.

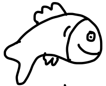


→ Slower than log n. Faster than n

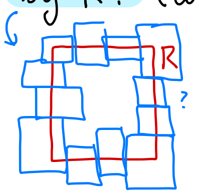
Stabbing: 3 cases

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

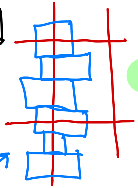




How many cells are stabbed by R? (worst case)



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



Analysis: How efficient is our algorithm?

→ Tricky to analyze

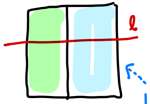
→ At some nodes we recurse on both children ⇒ $O(n)$ time?

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

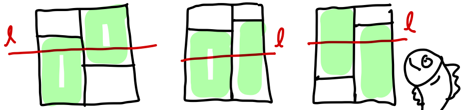
$\Rightarrow T(n) = n^{\log_b a}$

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

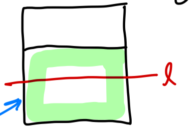
Since tree is balanced a child has half the pts + grandchild has quarter.

Recurrence: $T(n) = \underbrace{2}_{\text{2 cells stabbed}} + \underbrace{2}_{\text{Recurse on 2 grandchildren}}T(\underbrace{n/4}_{\text{Each has n/4 pts}})$

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



p splits horizontally



ℓ stabs only one

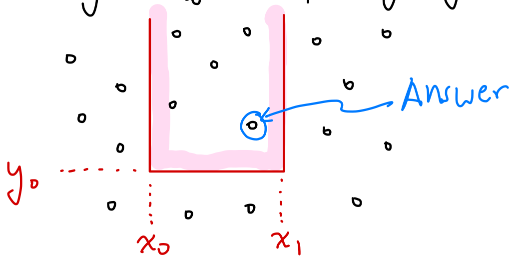
Range Tree Applications:

- 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

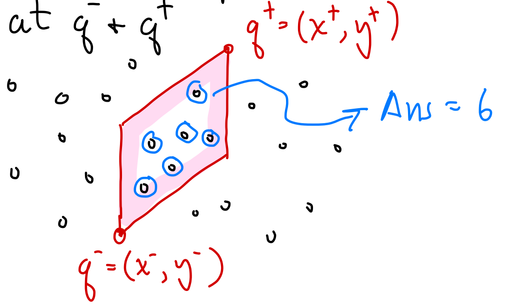
Given a set P of n pts in $\mathbb{R}^2$, a query consists of x-interval $[x_0, x_1]$ and y value y_0. Return the lowest pt in 3-sided region $x_0 \le x \le x_1$ + $y \ge y_0$



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^+

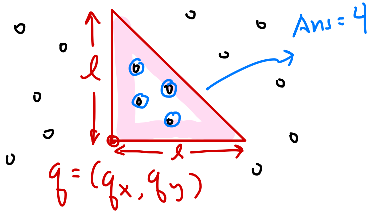


Return a count of the number of pts of P inside the skewed rectangle.

Adding New Coordinates:

NE Right Triangle Query

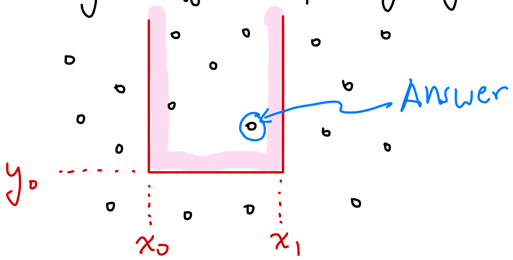
Given a set P of n pts in $\mathbb{R}^2$ and scalar $\ell > 0$, a NE triangle is a 45-45 right triang with lower left corner at q and side length ℓ.



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

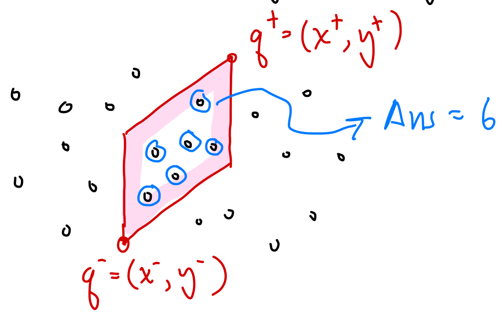
Query Processing:

- 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.

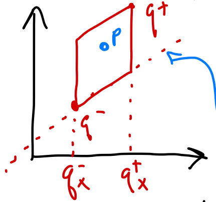
Analysis:

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

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

$\iff q_y^- - q_x^- \le p_y - p_x \le q_y^+ - q_x^+$

$p_y' = p_y - p_x$

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

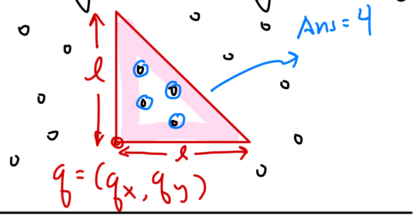
Let P' be resulting set.

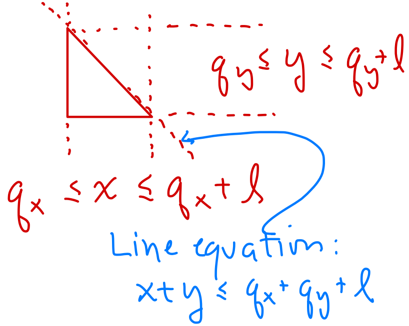
Build std. range tree for P'. Return ans. to query

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

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

NE Right Triangle Query





- 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

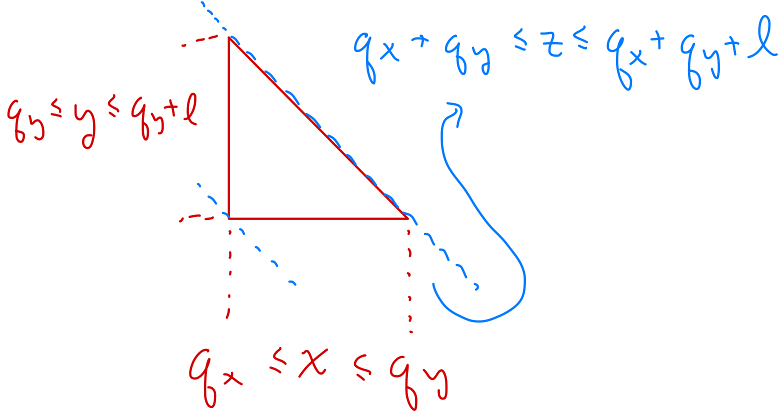
Build a 3D range tree on P'

NE triangle query becomes:

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

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

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



Space:

$O(n \log^2 n)$

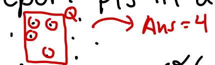
Query time:

$O(\log^3 n)$

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:** $O(k + \log^d n)$

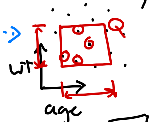
→ In $\mathbb{R}^2$: $\log^2 n$ **much better** than $\sqrt{n}$ for large n

→ Range trees are **more limited**

Layering:

Combining search structures

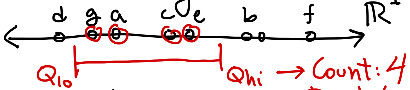
- Suppose you want to answer a **composite query** w. multiple criteria:
- Medical data: Count subjects w/



  - **Age range:** $a_{lo} \leq age \leq a_{hi}$
  - **Weight range:** $w_{lo} \leq weight \leq w_{hi}$
- Design a data structure for each criterion **individually**
- **Layer** these structures together to answer full query

→ **Multi-Layer Data Structures**

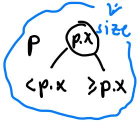
1-Dim Range Tree:



→ Count: 4
Rept: {g,a,c,e}

Approach:

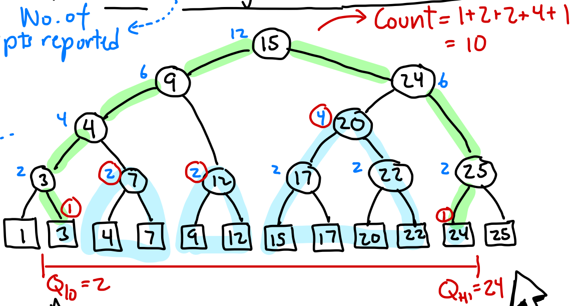
- Balanced BST (eg. AVL, RB,...)
- Assume **extended tree**
- Each node p stores no. of entrie in subtree: **p.size**



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 $O(k + \log n)$)

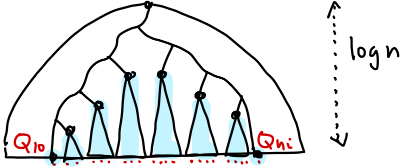
No. of pts reported



→ Count = 1+2+2+4+1 = 10

Canonical Subsets:

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

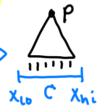


Range Trees II

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)

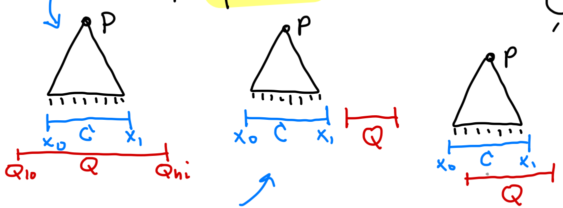
Cases:

p is external:

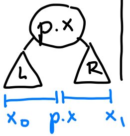
- if $p.pt.x \in Q \rightarrow 1$ else $\rightarrow 0$

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)
    return $p.pt.x \in Q$   T → 1, F → 0
  else if ($C \subseteq Q$) 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, $x_1$])
```

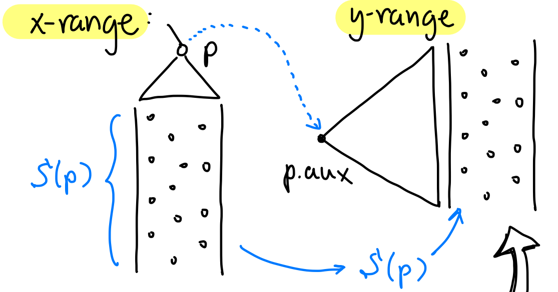
Analysis:

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)

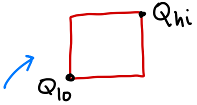
2-D Range Searching:

- "**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 $S'(p)$



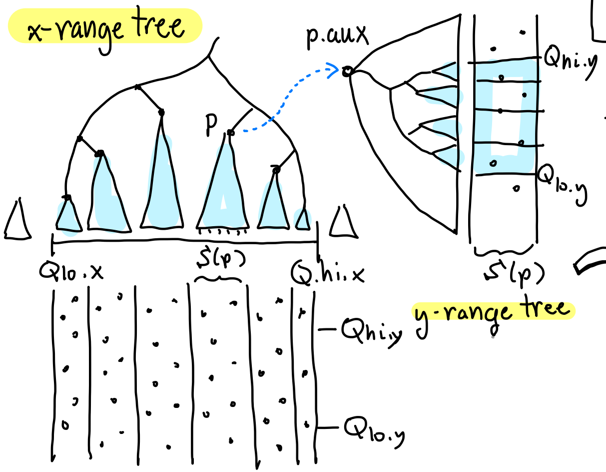
Range Trees III

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 range1Dy 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

⇒ 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? T→ 1  F→ 0
    else if (Q.x contains C){    // C ⊆ Q's x-projection
        [$y_0$,$y_1$] = [-∞,+∞]    // 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

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

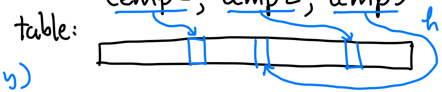
Overview:

- 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: temp1, temp2, temp3



Common Examples:

Assume keys can be interpreted as ints

- 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

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

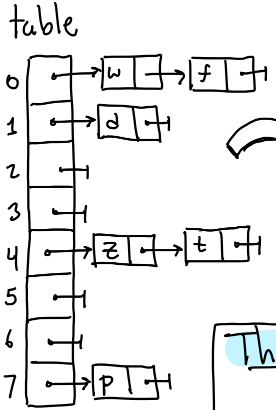
Separate Chaining:

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

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

success: 1 for head + half the list

unsuccess: 1 " " + all the list

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

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

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

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

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

Token-based – See latex notes!

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 $\lambda \le 1$
- To insert key:

check: $h(x)$ if not empty try
$h(x)+i_1$
$h(x)+i_2$
⋮

$\langle i_1, i_2, i_3, \dots \rangle$ – Probe sequence

- What's the best probe sequence?

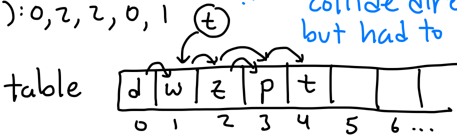
Linear Probing:

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

until finding first available

Simple, but is it good?

x: d, z, p, w, t
h(x): 0, 2, 2, 0, 1



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

Analysis:

Let S_{LP} = expected time for successful search
U_{LP} = " " unsuccessful "

Thm:

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

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

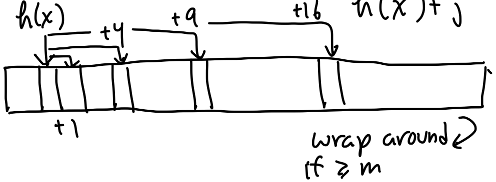
Obs: As $\lambda \to 1$ times increase rapidly

Clustering

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

Quadratic Probing:

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



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)

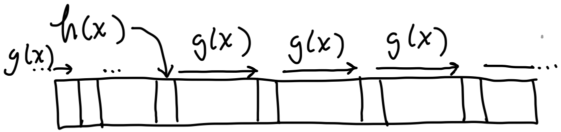
Open Addressing:
- Linear probing: } clustering
- Quadratic probing: } clustering

Double Hashing:

(Best of the open-addressing methods)

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

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



(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

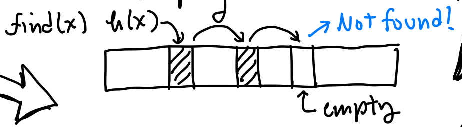
Very efficient!

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



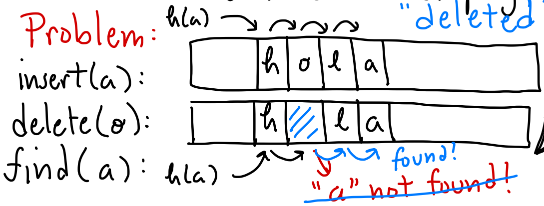
Delete(x): Apply find(x)

→ Not found ⇒ error

→ Found ⇒ set to ~~"empty"~~ "deleted"

~~Is this right??~~

Problem:

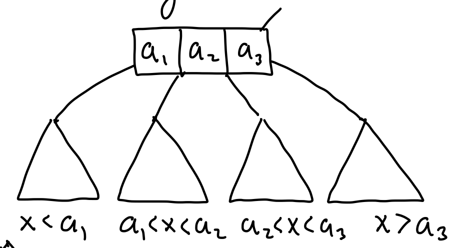


B-Trees I

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:

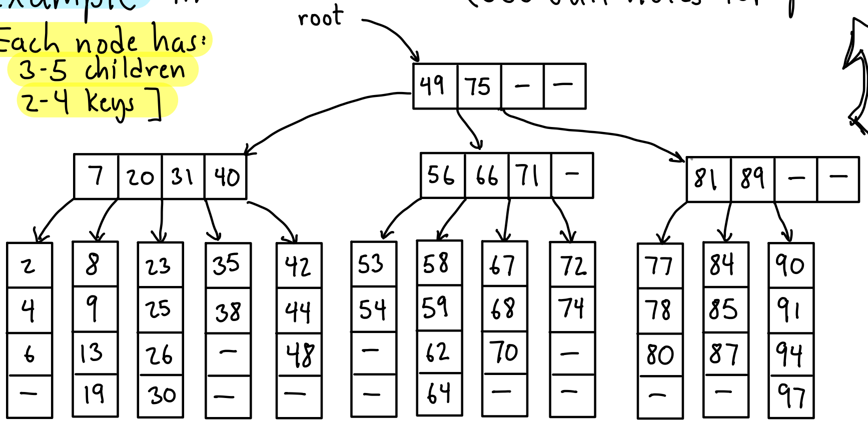
- Perhaps the most widely used search tree
- 1970 - Bayer + McCreight
- Databases
- Numerous variants

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 $\Rightarrow$ $k-1$ 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
}
```

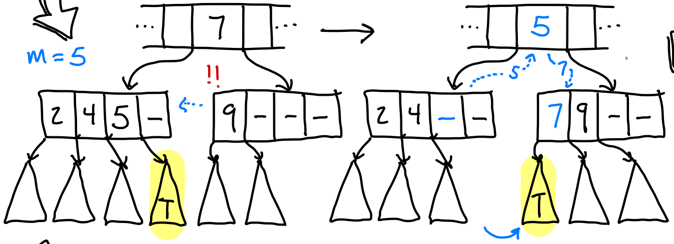
B-Trees II

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



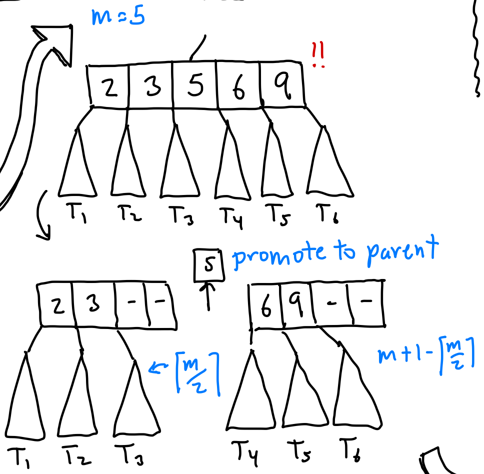
Node Splitting:

- 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



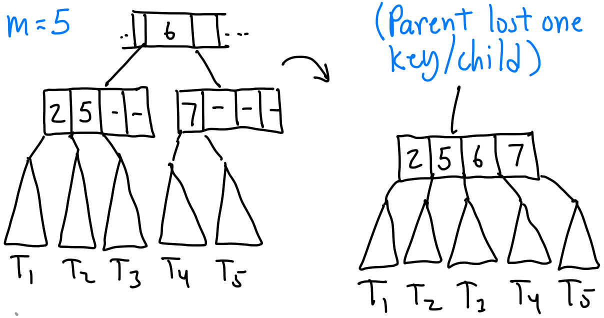
Node Merging:

- 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

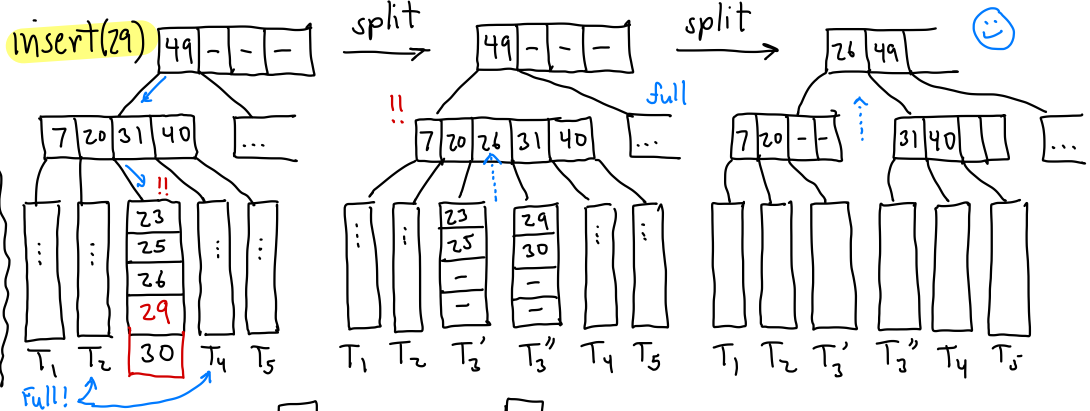


B-Trees III

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

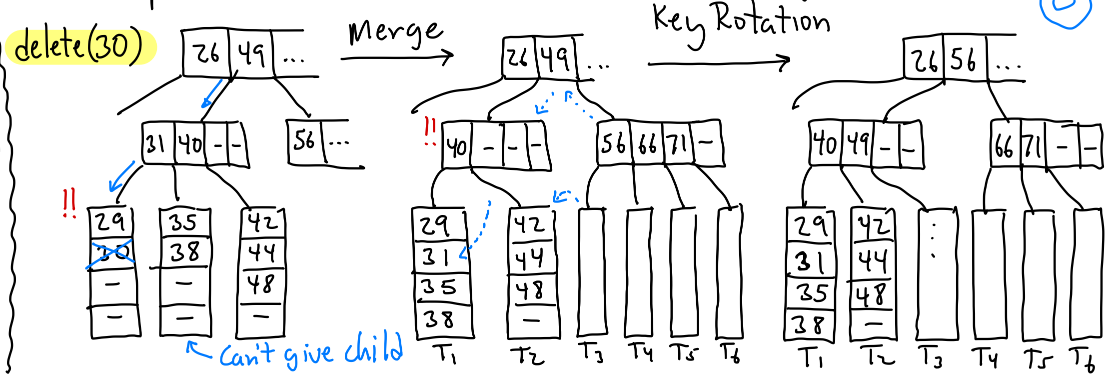
Example: m=5



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

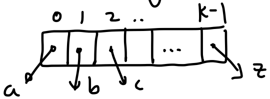


Tries and Digital Search Trees I

Tries: History

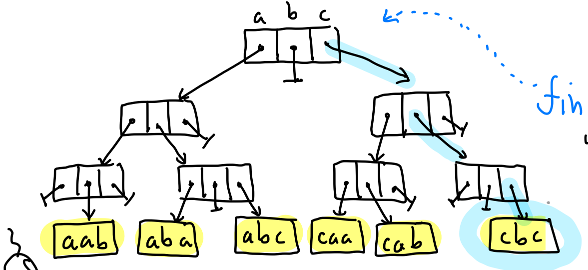
- de la Briandais (1959)
- Fredkin - "trie" from "retrieval"
- Pronounced like "try"

Node: Multiway of order k

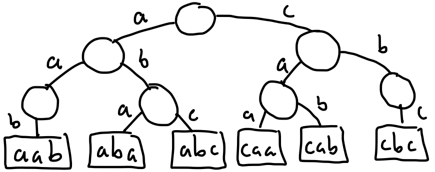


Example: $\Sigma = \{a=0, b=1, c=2\}$

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

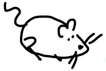


Same structure/Alt. Drawing



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: $a=0$, $b=1$, ... $z=k-1$



Analysis:

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

Space:

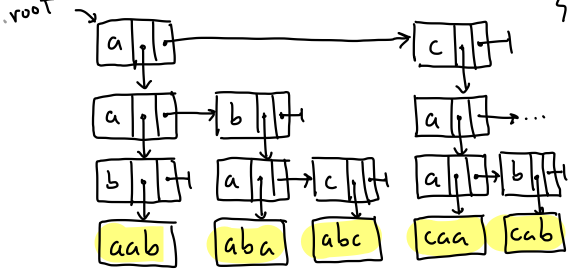
Large?

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

Analysis:

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

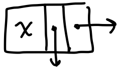
Example:



How to save space?

de la Briandais trees:

- Store 1 char. per node
-



≠ x ⇒ try next char in Σ

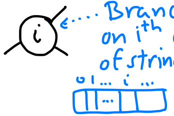
== x ⇒ advance to next character of search string

- First-child/ next-sibling

Tries and Digital Search Trees II

Patricia Tries:

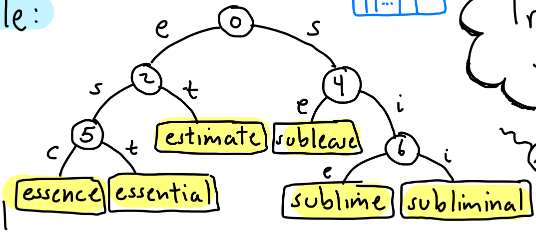
- Improves trie by compressing degenerate paths
- PATRICIA = Practical Alg. to Retrieve Info. Coded in Alpha...
- Late 1960's: Morrison + Gwehenberger
- Each node has index field, indicates which char to check next (Increase with depth)



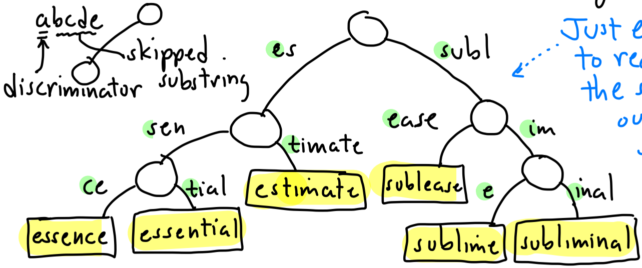
Branch based on i^{th} char of string

Example:

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



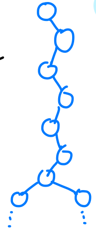
Same data structure – Drawn differently



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!

Example:

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

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·(No. of nodes) + (Storage for strings)

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 CMSC420 notes"

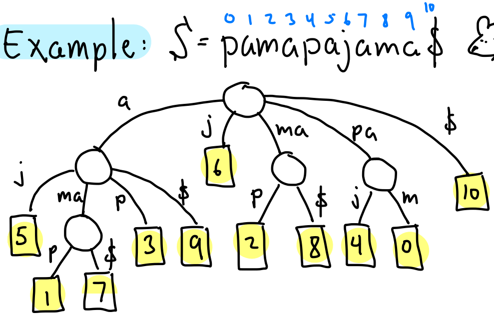
Notation: $S = a_0 a_1 a_2 \ldots a_{n-1}\$$

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

Example: S = pamapajama\$



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

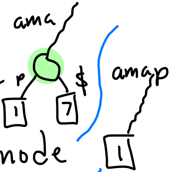
Example:

Search("ama") → End at intern node

Report: 2occ's.

Search("amapaj") → End at extern node

Goto S_1 + verify



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

Analysis:

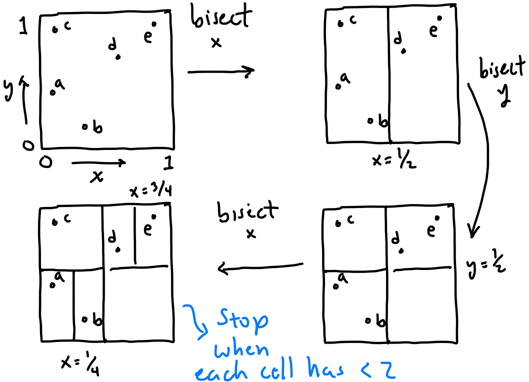
- **Space:** O(n) nodes
  O(n·k) total space
  (k = |Σ| = O(1))
- **Search time:** ~ total length of target string
- **Construction time:**
  - O(n·k) [nontrivial]

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

Geometric Applications:

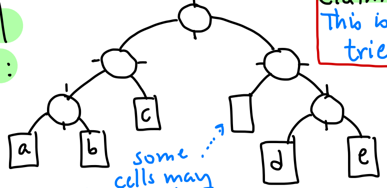
PR kd-Tree: kd-tree based on midpoint subdivision

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

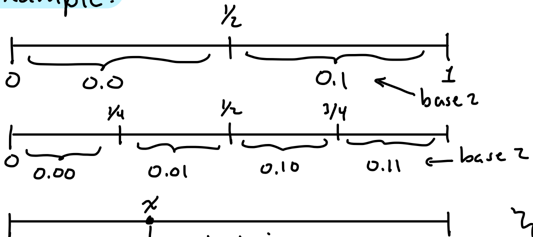
Binary Encoding:

- 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:

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

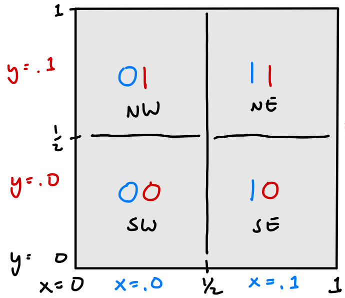
$$x = \sum_i a_i \cdot 1/2^i$$

Example:

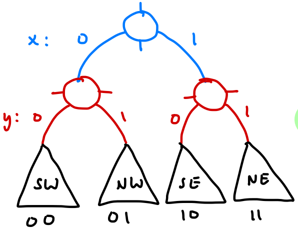


↳ Bit string: 010110110001...

How do we extend to 2-D?



PR kd-tree



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: $x = 0.a_1a_2\ldots$ in binary

$y = 0.b_1b_2\ldots$

Define:

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

Called **Morton Code** of p

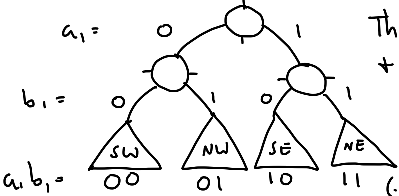
Further Remarks:

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

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

Lemma: Given a pt set $P \subseteq \mathbb{R}^2$ (in unit square $[0,1)^2$) let $P = \{p_1,\ldots,p_n\}$ where $p_i = (x_i,y_i)$
Let $\Phi(P) = \{\phi(p_1), \phi(p_2),\ldots,\phi(p_n)\}$
(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 $x = 0.a_1a_2\ldots$ $y = 0.b_1b_2\ldots$
and consider just $\phi(x,y) = a_1b_1\ldots$



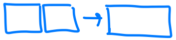
The PR kd-tree + binary trie assign pts to same subtrees.
(.... induction)

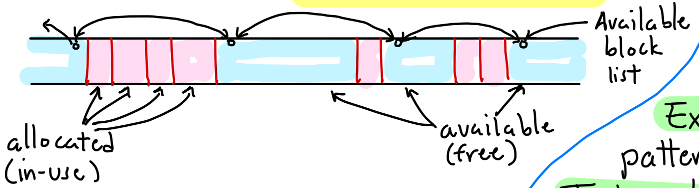
Memory Management I

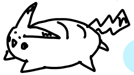
Deallocation Models:

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

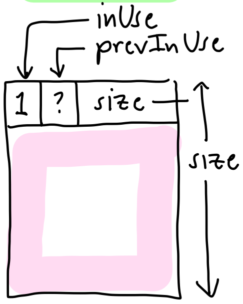
Fragmentation:
- Results from repeated allocation + deallocation (Swiss-cheese effect)

External: Caused by pattern of alloc/dealloc

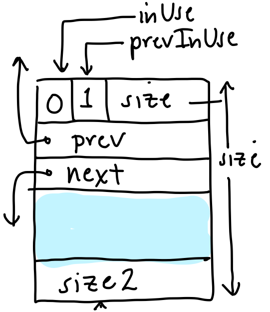
Internal: Induced by mem. manage. policies (not user)

Block Structure:

Allocated:



Available:



Also stores size

Guide:

prevInUse: 1 if prev. contig. block is allocated

prev/next: links in avail. list

size/size2: total block size (includes headers)

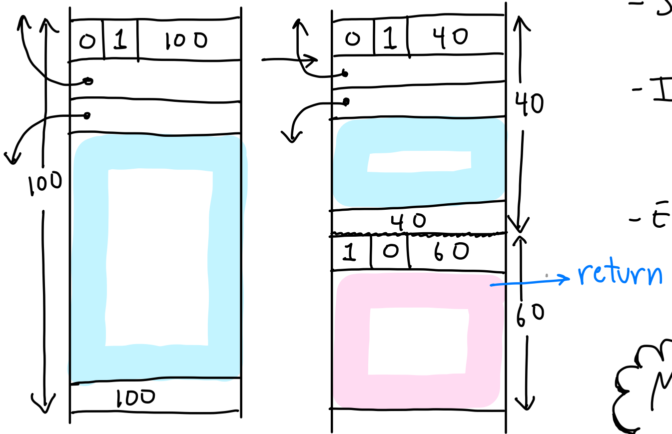
How to Select from available blocks?

- **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



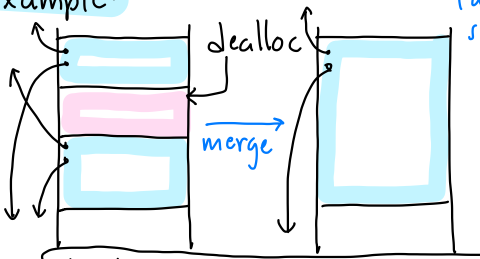
Allocation: malloc(b)

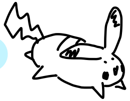
- Search avail. list for block of size b′ ≥ b+1
- If b′ close to b : alloc entire block (unlink from avail list)
- Else : Split block

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)
```

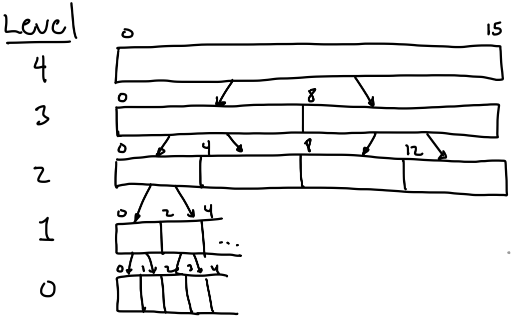
```
else {
    p.size -= b              // remove allocation
    *(p+p.size-1) = p.size   // size2
    q = p + p.size           // start of new block
    q.size = b               } // new block
    q.prevInUse = 0          }    header
}

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:

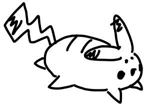


In practice: There is a minimum allowed block size

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.



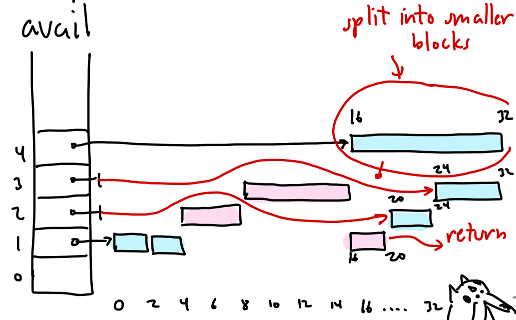
Merging:

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

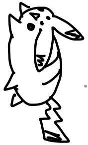
$\equiv \text{buddy}_k(x) = (1 \ll k) \oplus x$ [Bit manipulation]

Example: alloc(2) $\xrightarrow{\text{round up}}$ 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
size2 } not needed