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
- ....

**Course Overview:**
- Fundamental data structures + algorithms
- Mathematical techniques for analyzing them
- Implementation

Introduction to Data Structures
- Elements of data structures
- Our approach
- Short review of asymptotics

**Common:**
$O(1)$: constant time ☺ [Hash map]
$O(\log n)$: log-time (good) [Binary search]
$O(n^p)$: $p$ = constant: poly time
$O(\sqrt{n})$

**Asymptotic: "Big-O"**
- Ignore constants
- Focus on large $n$

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

**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 operations performed?

Our approach:
- **Theoretical**: Algorithms + Asymptotic Analysis
- **Practical**: Implementation + practical efficiency

**Asymptotic Analysis:**
- Run time as function of $n$: no. of items
- Worst-case, average case, randomized, ...
- **Amortized** - average over series of ops.

# Basic Data Structures I

- ADTs
- Lists, Stacks, Queues
- Sequential Allocation

## Abstract Data Type (ADT)

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

## Linear List ADT:

Stores a sequence of elements $\langle a_1, a_2, \ldots, a_n \rangle$. 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 → $a_1$ → $a_2$ → ... → $a_n$ → null

**Doubly:** head → $a_1$ ⇄ $a_2$ ⇄ ... ⇄ $a_n$, tail

Performance varies with implementation

## Doubling Reallocation:

When array of size n overflows

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

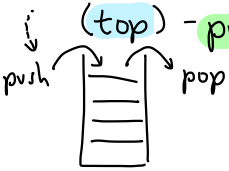
## Dynamic Lists + Sequential Allocation:

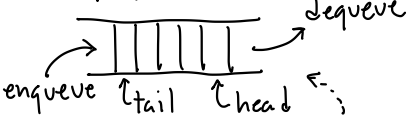
What to do when your array runs out of space?

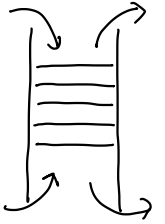
**Deque** ("deck"): Can insert or delete from either end

## Stack:

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



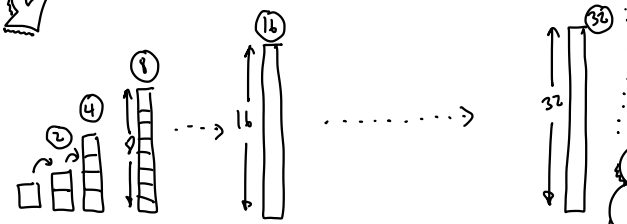



**Queue:** FIFO list: **enqueue** inserts at **tail** and **dequeue** deletes from **head**

## Cost model (Actual cost)

**Cheap:** No reallocation → 1 unit

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

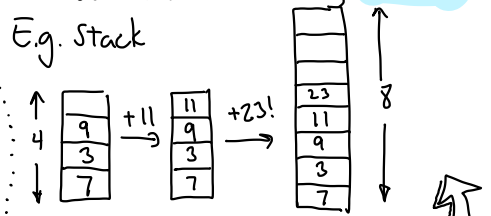



Total = 17+(2+4+8+16+32) = 79

## Dynamic (Sequential) Allocation

- When we overflow, double

E.g. Stack




## Basic Data Structures II

- Amortized analysis of dynamic stack

**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:

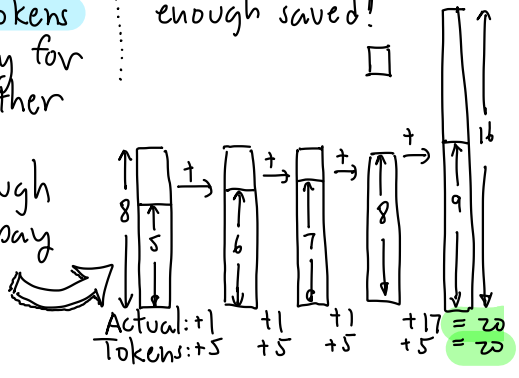
- Each request of push/pop we charge user 5 "work tokens"
- We use 1 token to pay for the operation + put other 4 in bank account.
- Will show there is enough in bank account to pay actual costs.

## 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! □



**Fixed Increment:** Increase by a fixed constant

$n \longrightarrow n + 100$

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

$n \longrightarrow 5 \cdot n$

**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?**

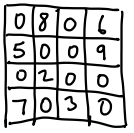
Leave as exercise ☹

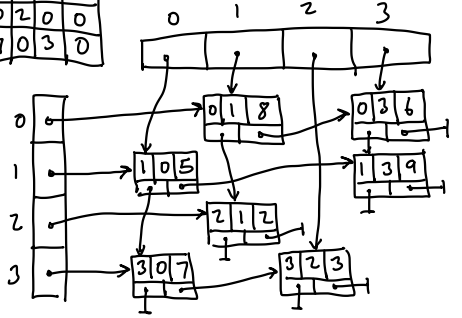
(Spoiler alert!)

- Fixed increment → no
- Fixed factor → yes
- Squaring → yes

**Dynamic Stack:**

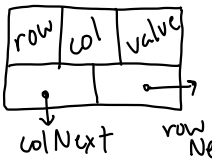
- Showed doubling ⇒ Amortized $O(1)$
- Other strategies?



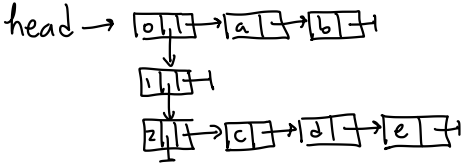

Basic Data Structures III
- Dynamic Stack - Wrap-up
- Multilists & Sparse Matrices

**Node:**



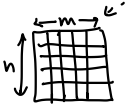

**Idea:** Store only nonzero entries linked by row and column

**Multilists:** Lists of lists




**Sparse Matrices:**

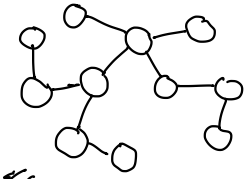
An $n \times m$ matrix has $n \cdot m$ entries and takes (naively) $O(n \cdot m)$ space



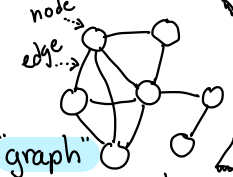

**Sparse matrix:** Most entries are zero

# Trees: Basic Concepts and Definitions

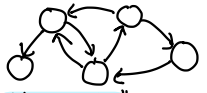
**Tree** (or "Free Tree")
- undirected
- connected
- acyclic graph



**Undirected**



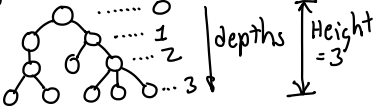

**Directed**

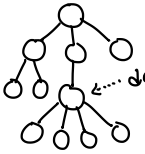



**Graph**: G = (V, E)
V = finite set of **vertices** (nodes)
E = set of **edges** (pairs of vertices)

**Depth**: path length from root

**Height**: (of tree) max depth

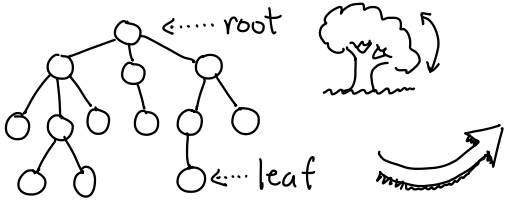



**Degree** (of node): number of children

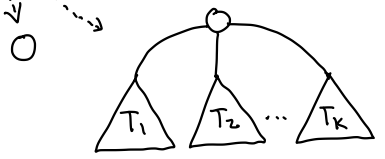
**Degree** (of tree): max. degree of any node

**Rooted tree**: A free tree with **root** node

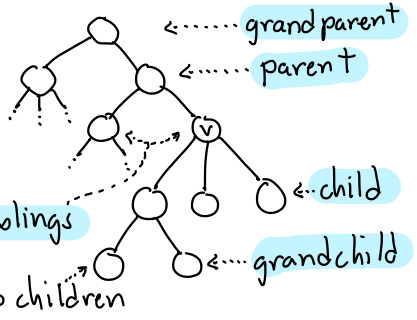



Formal definition:

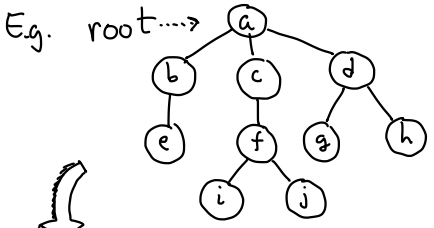
**Rooted tree**: is either
- single node (root)
- set of one or more rooted trees ("subtrees") joined to a common root

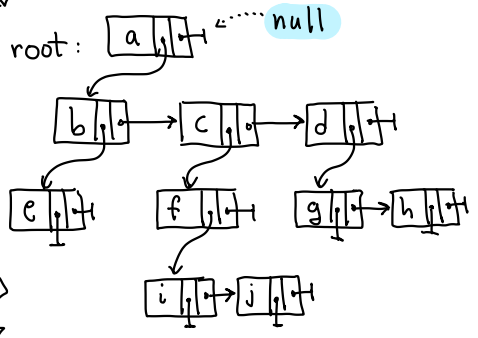



"Family" Relations




**Leaf**: no children



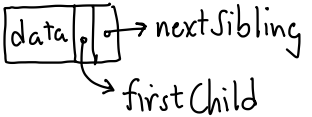




called the **Binary representation**

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

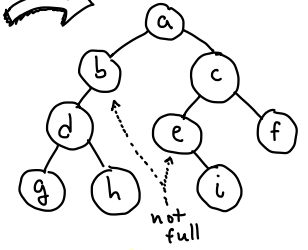
**Representing rooted trees**:
Each node stores a (linked) list of its children

**Node structure**:

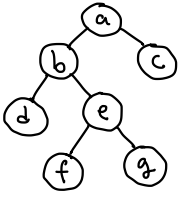



Trees
Representation
+ Binary Trees (1)

(Not full)




Full:

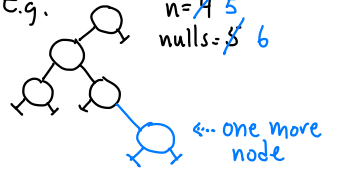



**Full**: Every non-leaf node has 2 children

**Wasted space?**

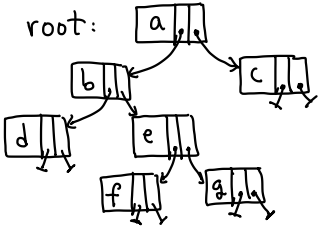
**Theorem**: A binary tree with n nodes has n+1 null links

E.g.

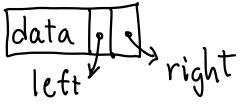



**In Java**: class BTNode<E> { ..... generic data

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



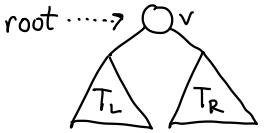

**Node structure**:



```
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
}
```

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

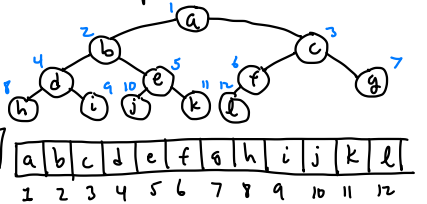
**Binary Tree Traversals** (can be generalized)




- process/visit v
- traverse $T_L$
- traverse $T_R$

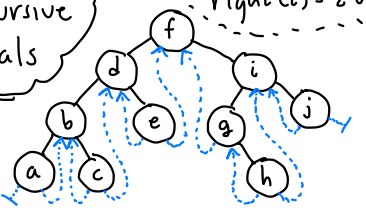
} recursive

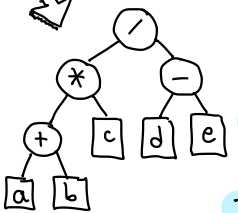
**Complete Binary Tree:** All levels full (except last)




parent(i) = $\lfloor i/2 \rfloor$
left(i) = 2·i
right(i) = 2·i+1

**Challenge:** Nonrecursive traversals






**Preorder:** / * + a b c - d e

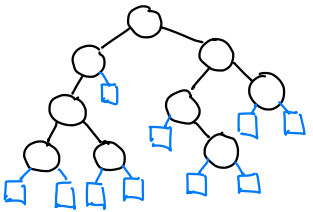
**Postorder:** a b + c * d e - /

**Inorder:** a + b * c / d - e

**Binary Trees: Traversals, Extension, and More**

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

**Dictionary:**

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

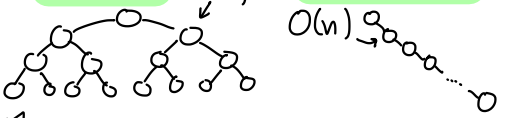
**delete** (Key x)
- delete x from dict. (Error if x not there)

**find** (Key x)
- returns a reference to associated value v, or **null** if not there.

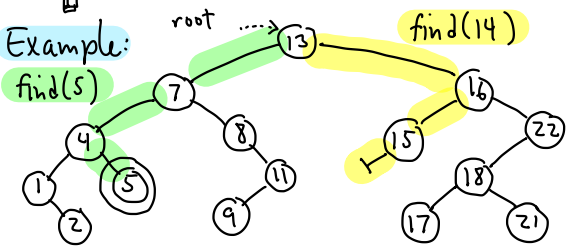
**Search:** Given a set of n entries each associated with **key** xi and **value** vi
- Store for quick access + updates
- **Ordered:** Assume that keys are totally ordered: <, >, ==

**Efficiency:** Depends on tree's height

**Balanced:** O(log n) **Unbalanced:** O(n)



**Example:** find(5) find(14)

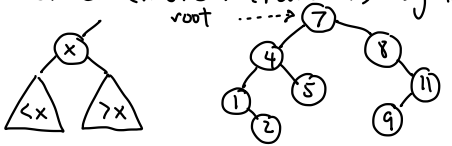



Binary Search Trees I
- Basic definitions
- Finding keys

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

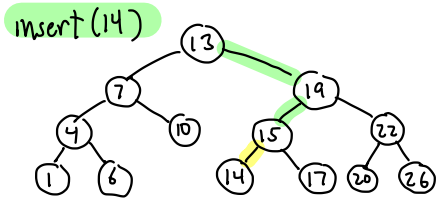
Can we achieve O(log n) time for all ops? ☆ Binary Search Trees ☆

**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
}
```

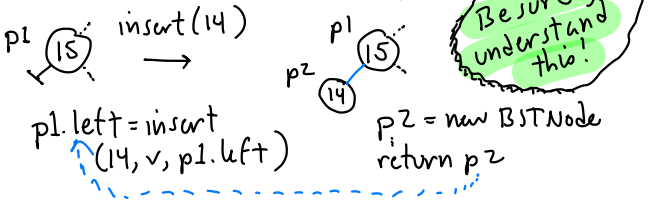



## Insert (Key x, Value v)

- 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)` ?



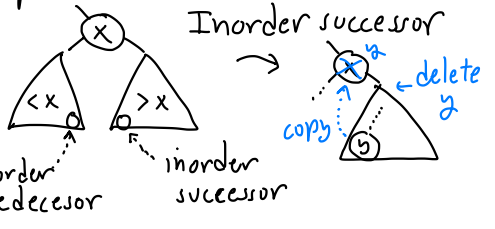

Binary Search Trees II
- insertion
- deletion

## Delete (Key x)

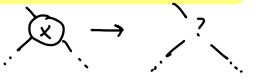
- find x
- if not found → error
- else: remove this node + restore BST structure

How?

## Replacement Node?



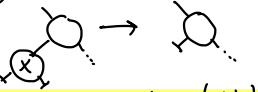

### 3. (x) has two children



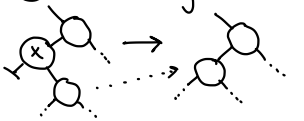

Find replacement node (y), copy to (x), and then delete (y)

### 3 cases:

① (x) is a leaf




② (x) has single child



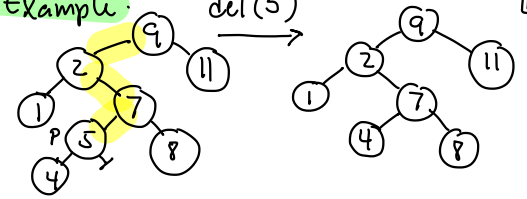
```
BSTNode delete(Key x, BSTNode p){
  if (p == null) Error! Key not found
  else
    if (x < p.key)
      p.left = delete(x, p.left)
    else if (x > p.key)
      p.right = delete(x, p.right)
    else if (either p.left or p.right null)
      if (p.left == null)
        return p.right
      if (p.right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

## Find Replacement Node

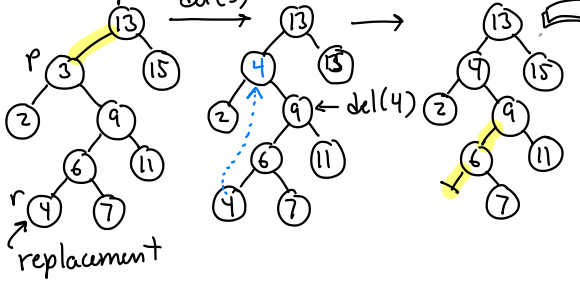
```
BSTNode findReplacement(BSTNode p){
  BSTNode r = p.right
  while (r.left ≠ null)
    r = r.left
  return r
}
```

Binary Search Trees III
- deletion
- analysis
- Java

Example: del(5)




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)

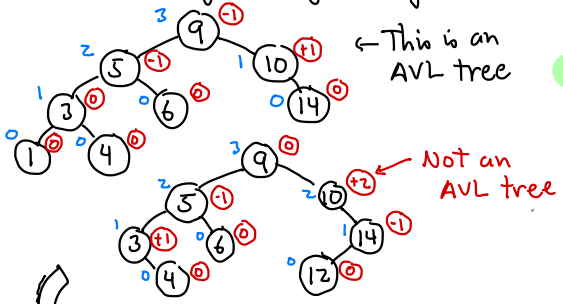
Local helpers (private or protected)

Data (private)

Public members (invoke helpers)

## Balance factor:

bal(v) = hgt(v.right) − hgt(v.left)



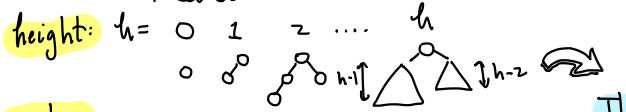

## AVL Height Balance

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

**AVL tree:** A binary search tree that satisfies this condition

## Does this imply O(log n) height?

Worst cases:




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 $F_{h+3} - 1$

AVL Trees I
- Basic defs
- Height props
- Rotations

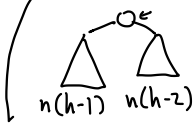
**Theorem:** An AVL tree of height h has at least $F_{h+3} - 1$ nodes.

**Proof:** (Induct. on h)

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

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

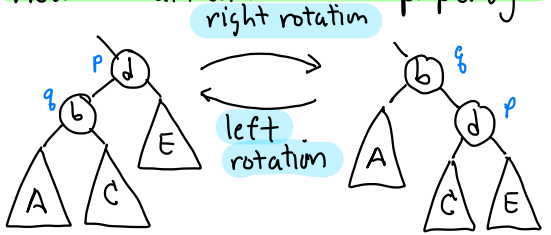
$h \geq 2$:




$$n(h) = 1 + n(h-1) + n(h-2)$$
$$= 1 + (F_{h+2} - 1) + (F_{h+1} - 1)$$
$$= (F_{h+2} + F_{h+1}) - 1 = F_{h+3} - 1 \quad \square$$

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

## How to maintain the AVL property?




**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"

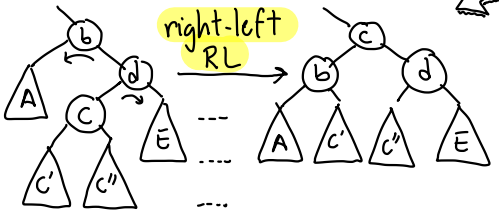
$$n \geq \varphi^{h+3} = c \cdot \varphi^h \Rightarrow h \leq \log_\varphi n + c'$$
$$\Rightarrow h \leq \log_2 n / \log_2 \varphi$$
$$= O(\log n) \quad \square$$

# AVL Trees II
- double rotations
- insertion

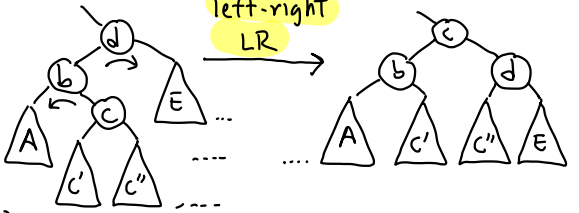
simpler than bal factor

## Double rotations:

right-left RL




left-right LR




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

## AVL Tree:

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

### Utilities:

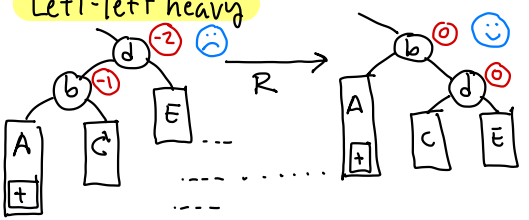
```
int height(AVLNode p)
    return { p == null --> -1
           { o.w.      --> p.height

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

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

### Find: Same as BST.

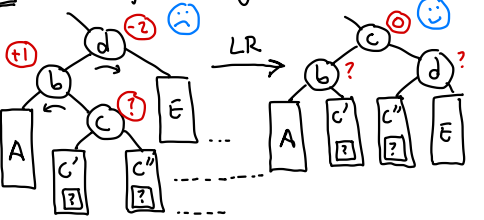
### Insert: Same as BST but as we "back out" rebalance

### How to rebalance? Bal = -2

**Left-left heavy**




**Left-right heavy:**



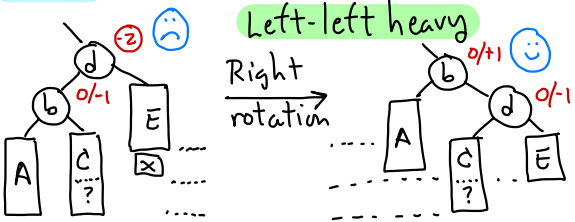

```
AVLNode rebalance(AVLNode p)
    if (p == null) return p
    if (balanceFactor(p) < -1)
        if (ht(p.left.left) >= ht(p.left.right))
            p = rotateRight(p)
        else p = rotateLeftRight(p)
    else if (balanceFact(p) > +1)
        ...(symmetrical)
    updateHeight(p); return p
```

```
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)
```

# AVL Trees III
- Deletion
- Examples

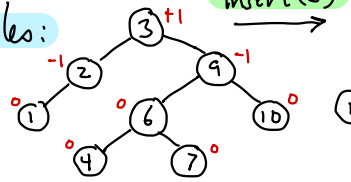
## Deletion: Basic plan
- Apply standard BST deletion
  - find key to delete
  - find replacement node
  - copy contents
  - delete replacement
- rebalance

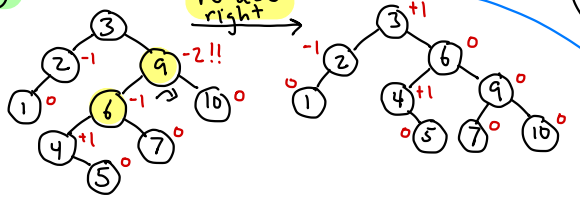
## Cases: Balance factor -2



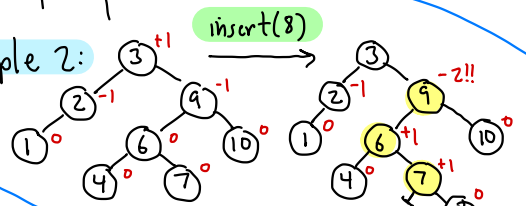

## AVLNode delete (Key x, AVLNode p)
same as BST delete
return rebalance(p)

## Examples:

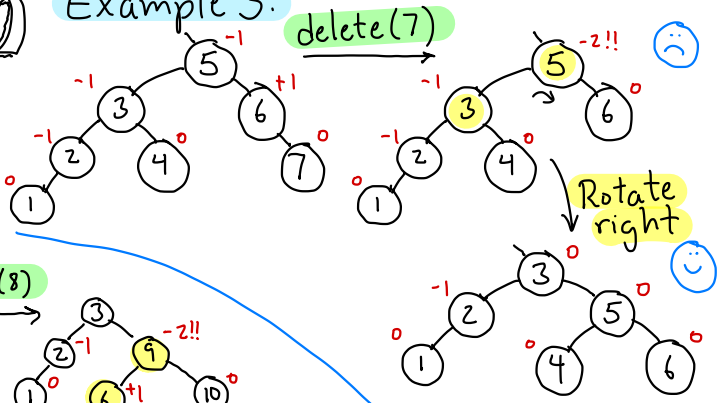


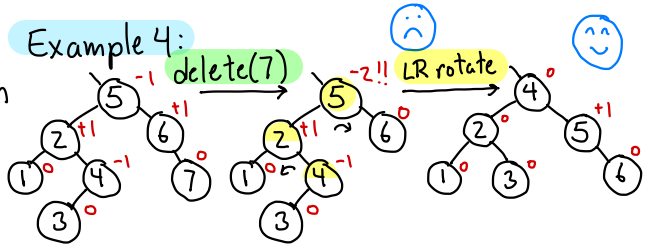

## Example 2:




## Example 3:

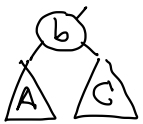



## Example 4:

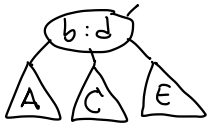


**Node types:**

**2-Node**
1 key
2 children



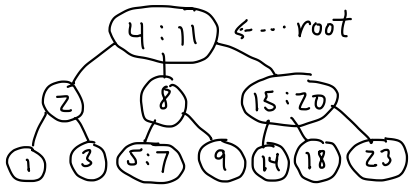

**3-Node**
2 keys
3 children




Identical heights

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

**Example:**
2-3 tree of height 2




**Recap:**

**AVL:** Height balanced Binary

**2-3 tree:** Height exact Variable width

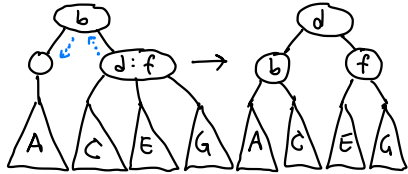



**Thm:** A 2-3 tree of $n$ nodes has height $O(\log n)$

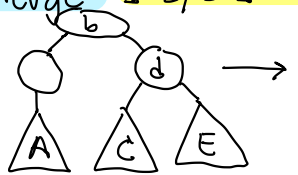
**Roughly:** $\log_3 n \leq h \leq \log_2 n$

**How to maintain balance?**
- Split
- Merge
- Adoption (Key rotation)

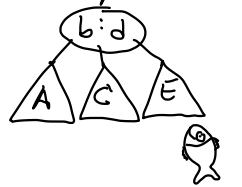
**Adoption (Key-Rotation)**
$1+3 = 2+2$




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

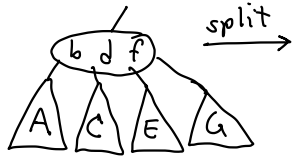



steal b from parent

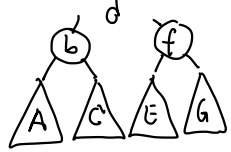





**Split:** $4 \rightarrow 2+2$



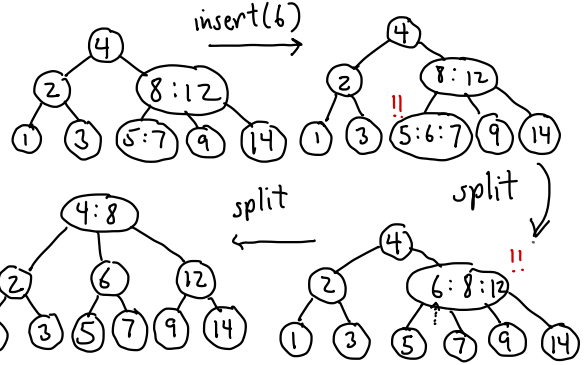

insert in parent




**Conceptual tool:**
We'll allow 1-nodes + 4-nodes temporary

# 2-3 Trees II

## Insertion example:




## Dictionary operations:

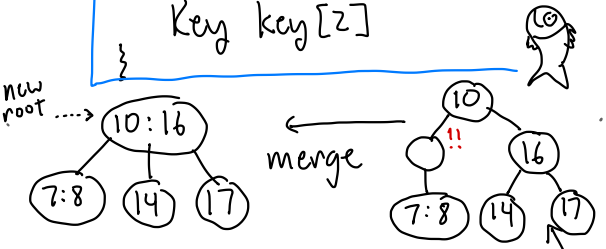
**Find** - straightforward

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

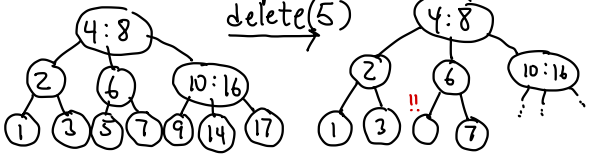
**Delete** - find / replacement / merge or adopt

## Implementation?

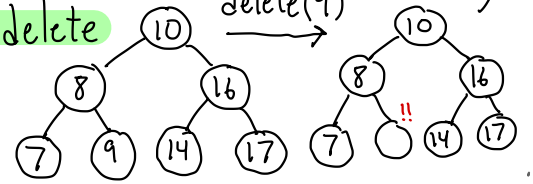
```
class TwoThree Node {
    int nChildren
    TwoThree Node children[3]
    Key key[2]
}
```




## Delete Example:



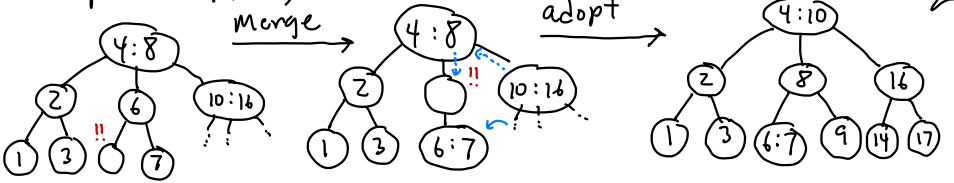

## Another delete example:



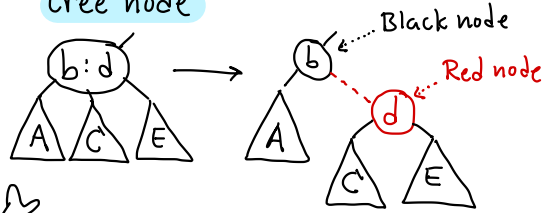

## Deletion remedy:

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

## Example (continued)

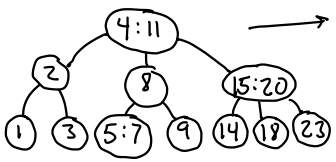


## Encoding 3-node as binary tree node

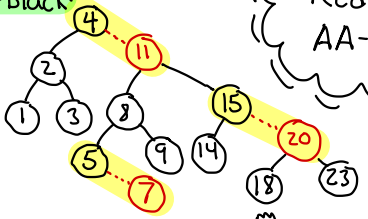



## Example:

2-3 Tree:




Red-Black:




## 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

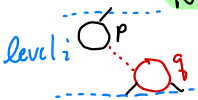
**Red-black Trees**: Guibas & Sedgewick 1978 (a binary variant of 2-3)

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

Red-Black and AA-Trees I

## 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:

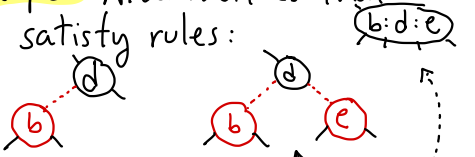
6. Each red node can arise only as right child (of a black node)

**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?

**Nope!** Alternatives that satisfy rules:



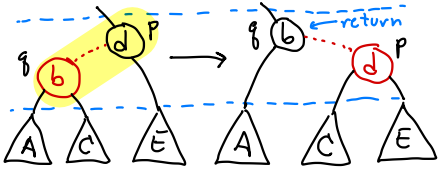

A "**left-skewed**" encoding

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

# Red-Black + AA Trees II

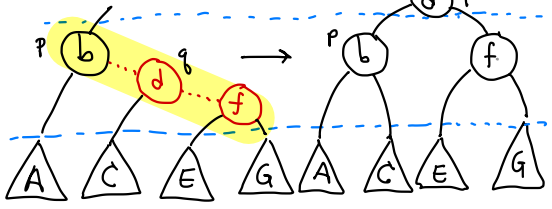
## Restructuring Ops:

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




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

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




What 2-3 op does this remind you of?

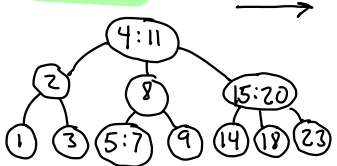
How to test?

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

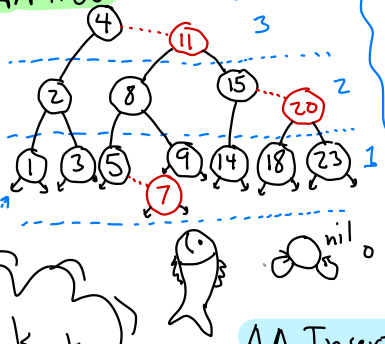
not needed (levels are monotone)

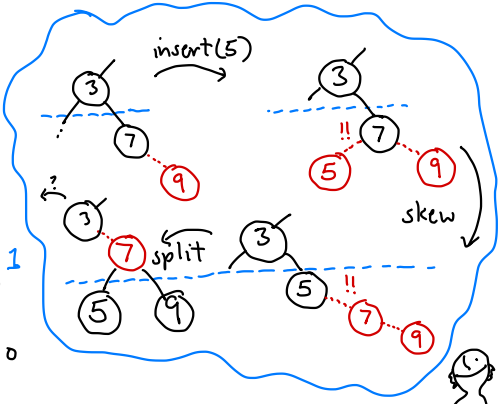
## Example:

2-3 Tree:




AA tree:



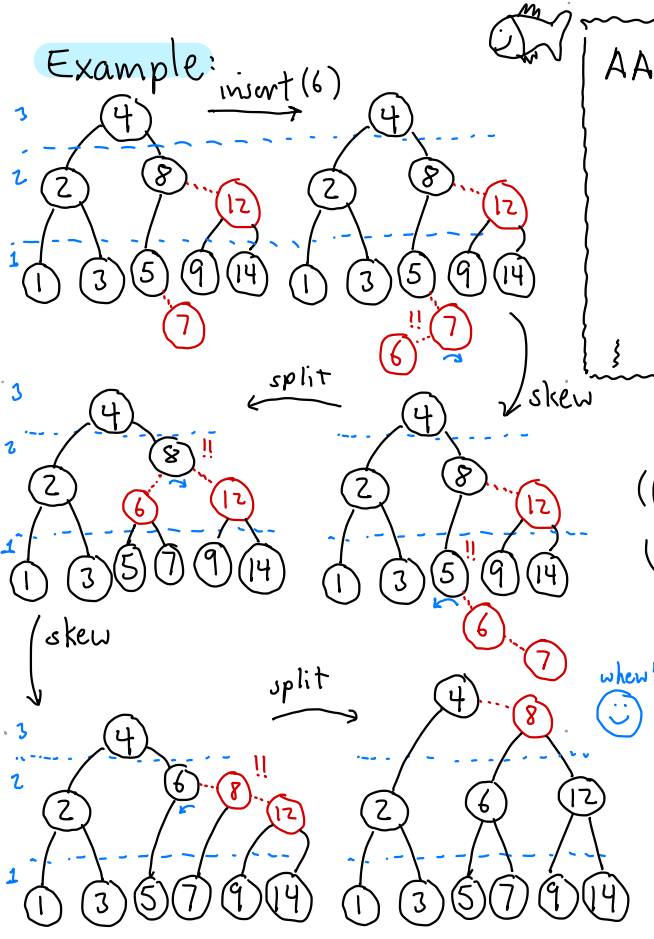

## AA Insertion:

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

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

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

# Red-Black and AA Trees III




```
AANode insert(Key x, Value v, AANode p){
  if (p == nil)                          // leaf level, left,right
    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))
}
```










**Deletion:**

Two more helpers:

**updateLevel:** If p's level exceeds $l = 1 + \min(p.left.level, p.right.level)$ 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

## History:

1989: Seidel & Aragon [Explosion of randomized algorithms]

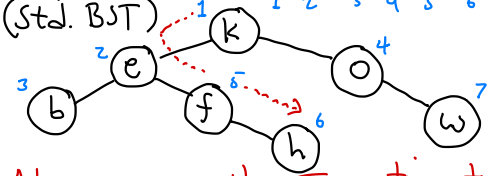
Later discovered this was already known: Priority Search Trees from different context (geometry) McCreight 1980

## Intuition:

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

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

(Std. BST)




Along any path - Insertion times increase

## Randomized Data Structures

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

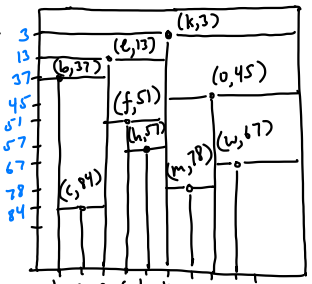
**Obs:** In a standard BST, keys are by inorder & insert times are in heap order (parent < child)

## Geometric Interpretation:

key → x

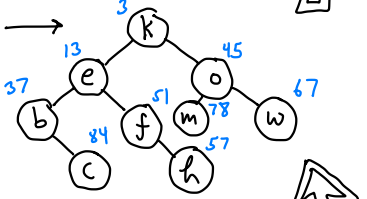
priority → y

McCreight's Priority Search Tree




**Example:**

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




**Treap:** Each node stores a key & a random priority.

Keys are in inorder.

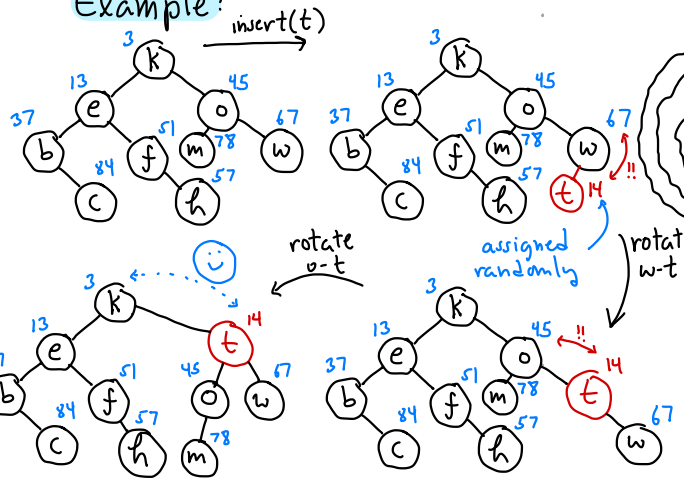
Priorities are in heap order

?Is it always possible to do both?

Yes: Just consider the corresponding BST

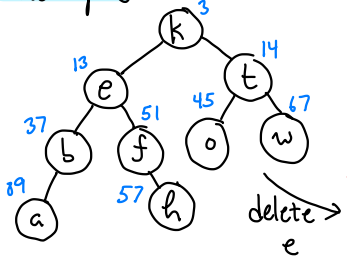
**Insertion:** As usual, find the leaf + create a new leaf node.
- Assign random priority
- On backing out - check heap order + rotate to fix.

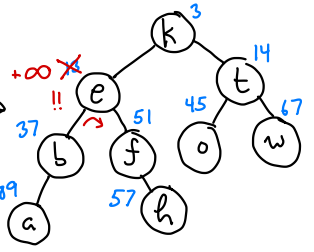
**Example:**




**Deletion:** (cute solution) Find node to delete. Set its priority to $+\infty$. Rotate it down to leaf level + unlink.

**Example:**




**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

Treaps II

**Implementation:** (See pdf notes)

**Node:** Stores priority + usual...

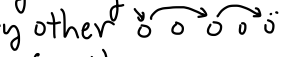
**Helpers:**

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


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

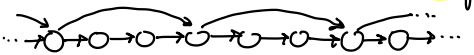
# Skip Lists I

**Ideal Skip List:**
- Organize list in levels
- Level 0: Everything
  1: Every other
  2: Every fourth
  $i$: Every $2^i$



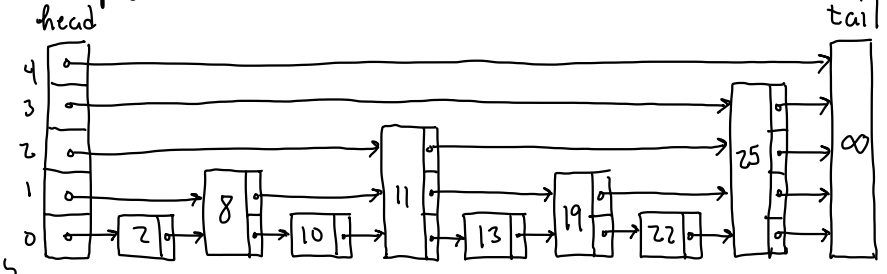
**Sorted linked lists:**
- Easy to code
- Easy to insert/delete
- Slow to search... $O(n)$

**Idea:** Add extra links to skip

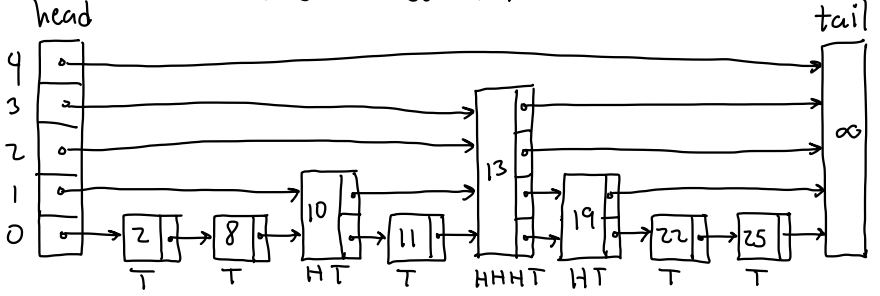


How to generalize?

**Example:**




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




**Node Structure:** (Variable sized)

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

In constructor, set level and size

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

current node

until we hit base level

advance horizontal

drop down a level

we are at base level

# Skip Lists II

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

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

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

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

→ Let $\ell = c \cdot \lg n$ ($\lg \equiv \log_2$)
Prob that max level exceeds $c \cdot \lg n$ is:

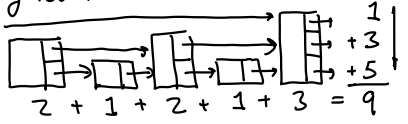
$$\leq n/2^\ell = n/2^{(c \cdot \lg n)} = n/(2^{\lg n})^c = n/n^c = 1/n^{c-1}$$

□

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

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

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




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

**Total space** (expected) is:

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

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

□

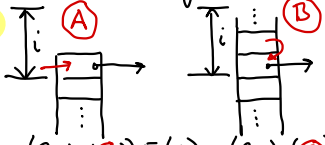
**Thm:** Expected search time is $O(\log n)$

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

**Obs** - Whenever search arrives first time to a node, it's at top level. (Can you see why?)

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

**Cases:**




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

$= 1 + 1/2 E(i) + 1/2 E(i-1)$

$\Rightarrow E(i)(1 - 1/2) = 1 + 1/2 E(i-1)$

$\Rightarrow E(i) = [1 + 1/2 E(i-1)]2 = 2 + E(i-1)$

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

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

⇒ We visit 2 nodes per level on average.

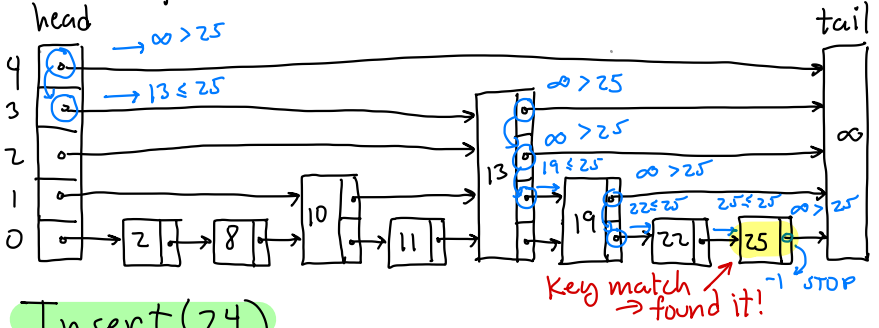
□

# Skip Lists III

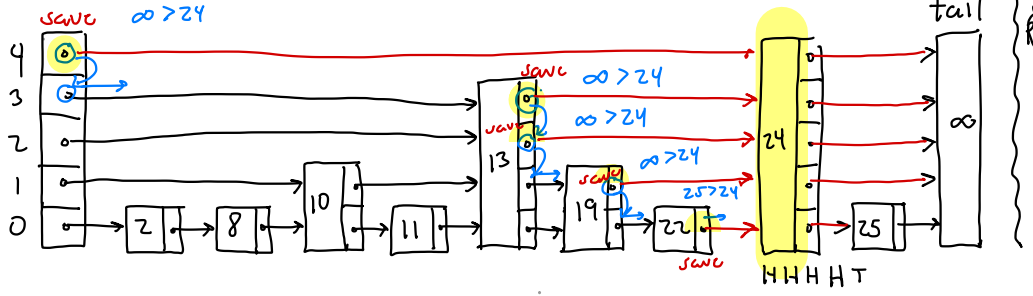
**Delete:**
- Start at top
- Search each level saving last node < key
- On reaching node at level 0, remove it and unlink from saved pointers

**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

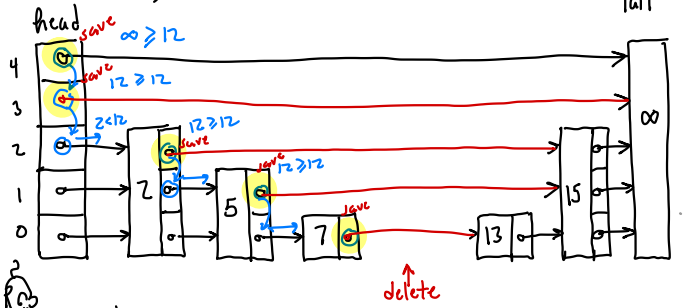
**Example:** find(25)




**Insert(24)**




**Delete(12)**




**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.

Other/Better Criteria?

Expected case: Some keys more popular than others

Self-adjusting: Tree adapts as popularity changes

How to design/analyze?

Recap: Lots of search trees
- Unbalanced BSTs
- AVL Trees
- 2-3, Red-black, AA Trees
- Treaps + Skip lists

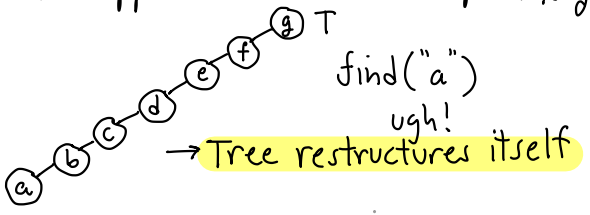
→ Focus: Worst-case or randomized expected case

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

SPLAY TREES I

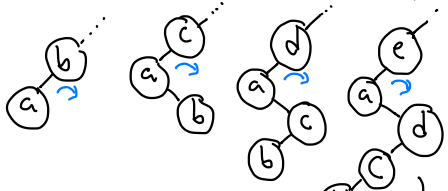
Splay Tree: A self-adjusting binary search tree
- No rules! (yay anarchy!)
  - No balance factors
  - No limits on tree height
  - No colors/levels/priorities
- Amortized efficiency:
  - Any single op - slow
  - Long series - efficient on avg.

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

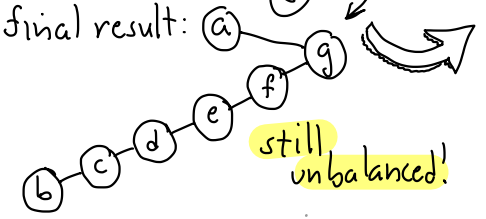



→ Tree restructures itself

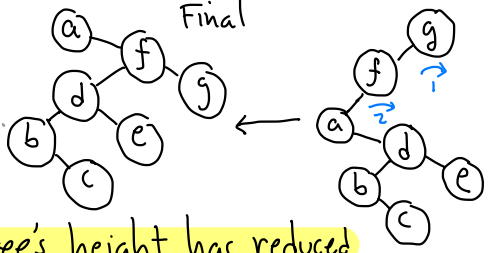
Idea I: Rotate "a" to top (Future accesses to "a" fast)



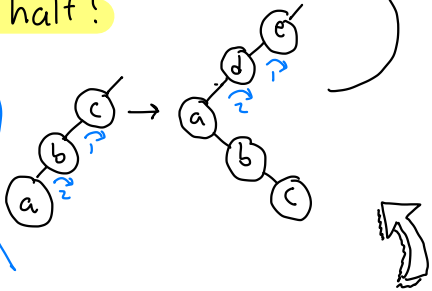
....final result:



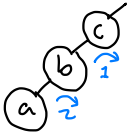

Final



Tree's height has reduced by ~ half!

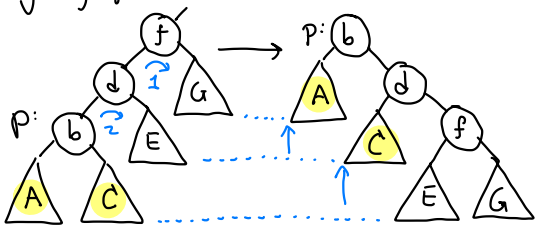


Idea II: Rotate 2 at a time - upper + lower



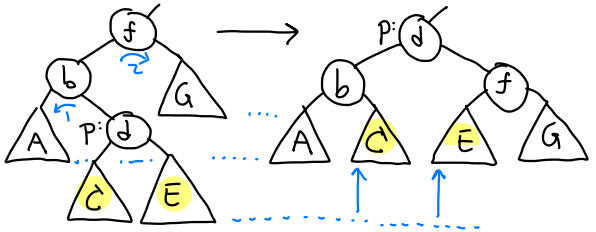
# Splay Trees II

## zigzig(p): [LL case]



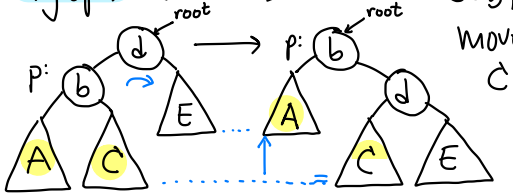

Subtrees A, C move up ↑

## zigzag(p): [LR case]




Subtrees C, E of p move up ↑

## zig(p): [L case]




Subtree A moves up ↑
C unchanged

## 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)
```

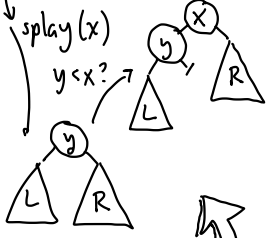
## insert(x):

```
splay(x)                          ...→ root.key ≠ x or error!
q = new Node(x)
if (root.key < x)
    x.left = root
    x.right = root.right
    root.right = null
else ... symmetrical...
```

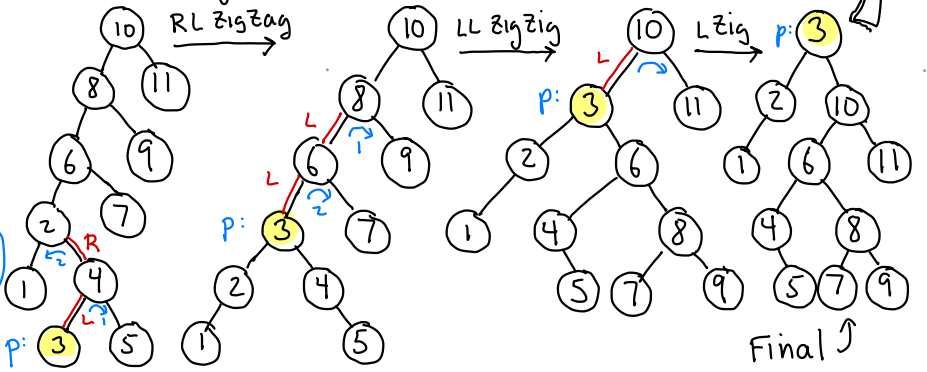
## find(x):

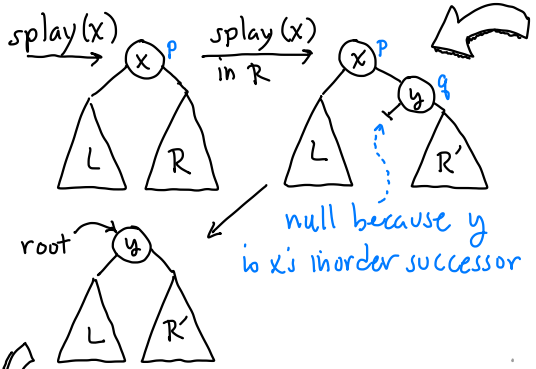
```
splay(x)
if (root.key == x)
    found!
else not found
```

splay(x)




## Example: splay(3)






**delete(x):**

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

**Dynamic Finger Theorem:**

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


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

# Splay Trees III

**Analysis:**

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

**Splay Trees are Amazingly Adaptive!**

**Balance Theorem:** Starting with an empty dictionary, any sequence of m accesses takes total time $O(m \log n + n \log n)$ where n = max. entries at any time.

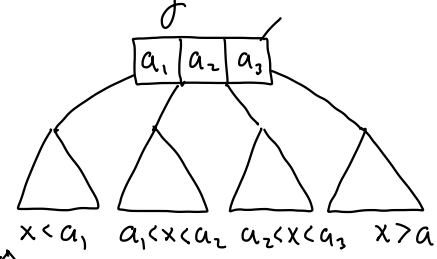
**Static Optimality:**

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

**Static Optimality Theorem:**

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

## Multiway Search Trees:




## B-Tree:

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

**B-Tree:** of order m ($\geq 3$)

- Root is leaf or has $\geq 2$ 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

## 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

B-Trees I

## Node Structure: constant int M=...

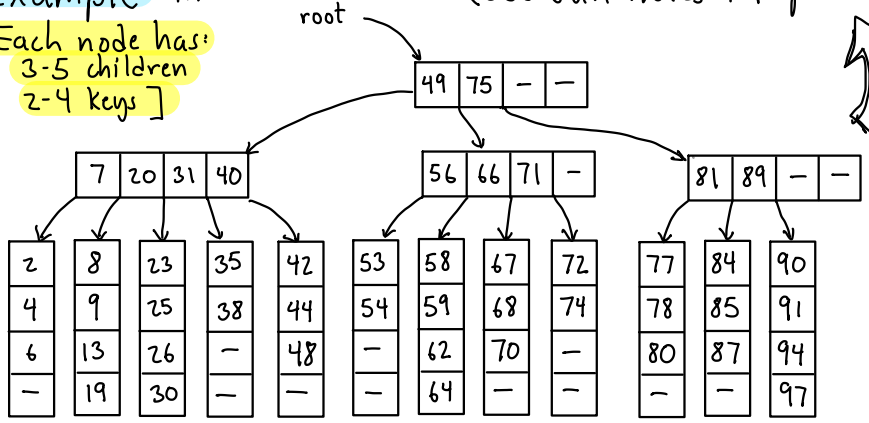
```
class BTreeNode {
    int nChild // no. of children
    BTreeNode child[M] // children
    Key key[M-1] // keys
    Value value[M-1] // values
}
```

**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)

## Example: m=5

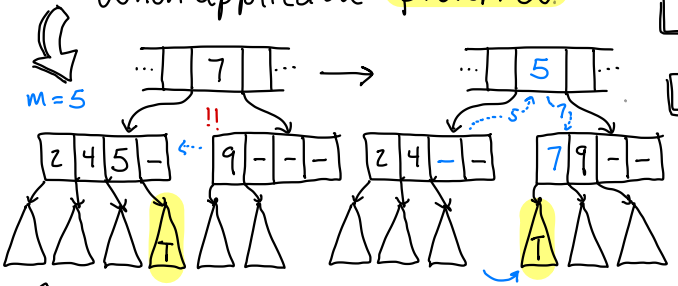
[Each node has:
3-5 children
2-4 keys]



# B-Trees II

## 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



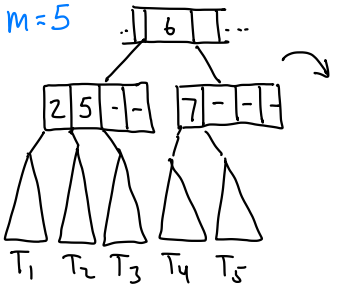

## Node Splitting:

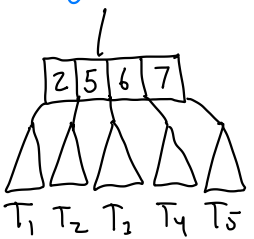
- After insertion, a node has too many children... $m+1$
- We split into two nodes of sizes $m' = \lceil m/2 \rceil$ and $m'' = m+1-\lceil m/2 \rceil$

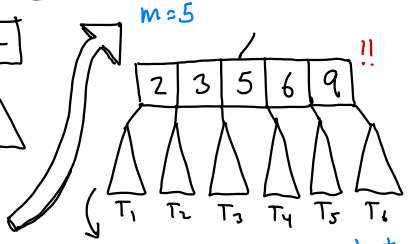
**Lemma:** For all $m \geq 2$,
$\lceil m/2 \rceil \leq m+1-\lceil m/2 \rceil \leq m$
$\Rightarrow$ $m'$ & $m''$ are valid node sizes

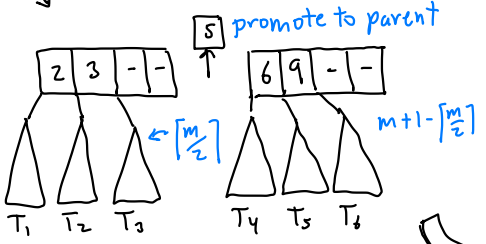
## B-Tree restructuring:

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




**Lemma:** For all $m \geq 2$,
$\lceil m/2 \rceil \leq 2\lceil m/2 \rceil - 1 \leq m$
$\Rightarrow$ Resulting node is valid

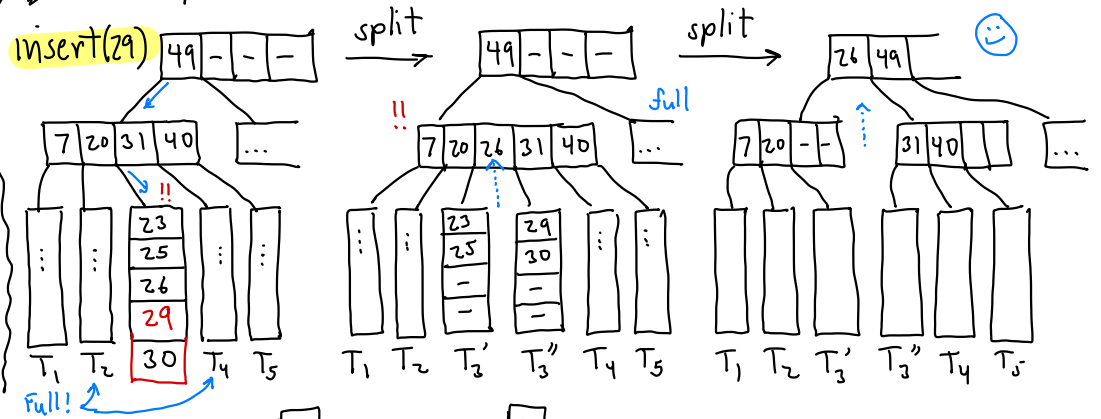
## Node Merging:

- 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

## 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

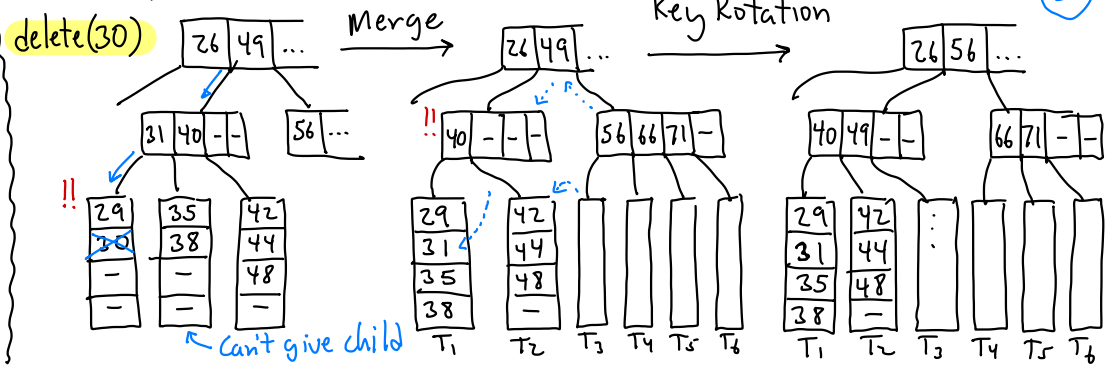



# B-Trees III

## Deletion:

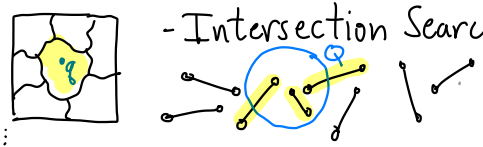
- Find key to delete
- Find replacement/copy
- If **underfull** ($\lceil m/2 \rceil - 1$) child
  - → If sibling can give child
    - → **Key rotation**
  - → Else (sibling has $\lceil m/2 \rceil$)
    - → **Merge** with sibling
  - → Propagates → If root has 1 child → collapse root

## Example: m=5



## Geometric Search:

- Nearest neighbors → P, q
- Range searching → R, 6 pts
- Point Location
- Intersection Search

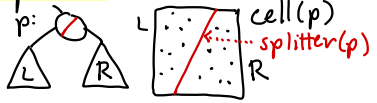



## So far: 1-dimensional keys

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

## Partition Trees:

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



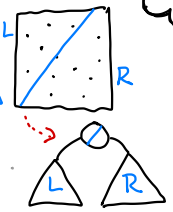

- External nodes store pts.

Quadtrees & kd-Trees I

## Multi-Dim vs. 1-dim Search?

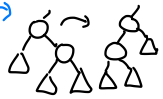
**Similarities:**

- 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:** $p = (p_1, \ldots, p_d)$ in real d-dim space $\mathbb{R}^d$
- **Other geom objects:** Built from these

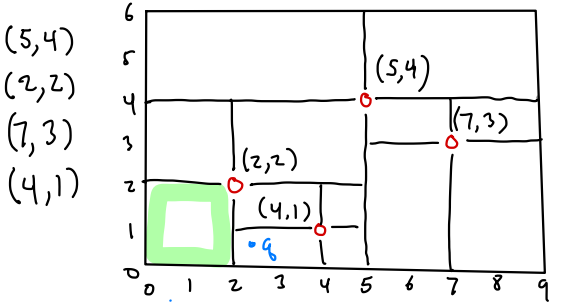
**Point:** A d-vector in $\mathbb{R}^d$

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

```
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...
}
```

## Point Quadtree:

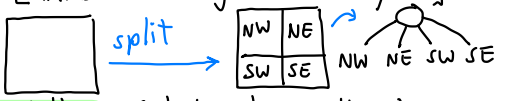
- Each internal node stores a point
- Cell is split by horiz. + vertic. lines through point




Each external node corresps. to cell of final subdivision

## Quadtrees (abstractly)

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




- Splitter: Subdivides cell into four (generally $2^d$) subcells

Quadtrees & kd-Trees II

## 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)

## 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

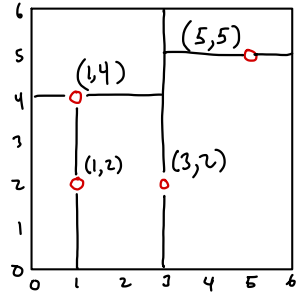
- splitter: Horiz. or vertic. line in 2-d (orthogonal plane o.w.)
- cell: Still AABB

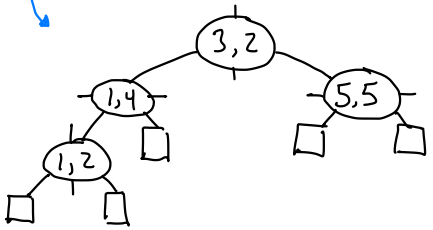

## Quadtrees - Analysis

- Numerous variants!
  PR, PMR, QR, QX, ... see Sameti 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?

## Example:

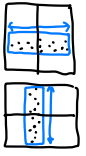

(3,2)
(1,4)
(5,5)
(1,2)




## How do we choose cutting dim?

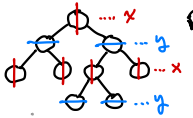
- **Standard kd-tree**: cycle through them (eg. d=3: 1,2,3,1,2,3...) based on tree depth
- **Optimized kd-tree**: (Bentley) → Based on widest dimension of pts in cell.



## Kd-Tree Node:

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

Quadtrees & Kd-Trees III



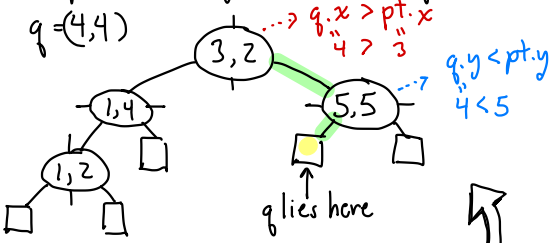

## Find:

- Descend the tree
- Compare query pt with node pt along cutDim

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

## Example: find(q) calls find(q, root)

q = (4,4)




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

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

```
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)
}
```

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

cutting dimension to use

dimension of point

## Kd-Tree Insertion:

(Similar to std. BSTs)

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


Quadtrees & kd-Trees IV

## Deletion:

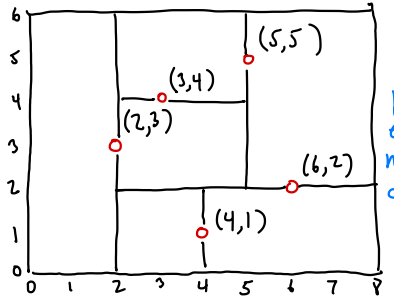
- Descend path to leaf
- If found:
  - leaf node → just remove
  - internal node
    - → find replacement
    - → copy here
    - → recur. delete replacement

This is the hardest part. See Latex notes.

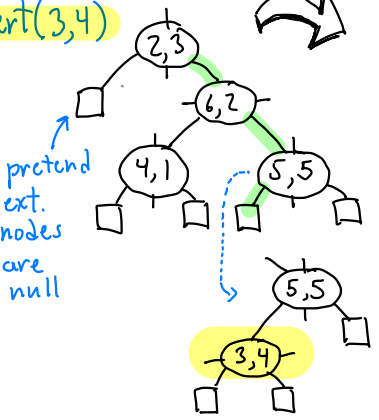
## Rebalance by Rebuilding:

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

## Example:




insert(3,4)




pretend ext. nodes are null

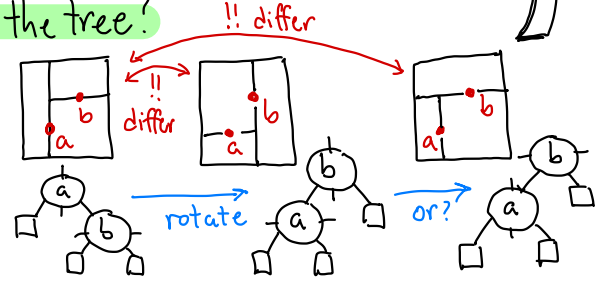
## Analysis:

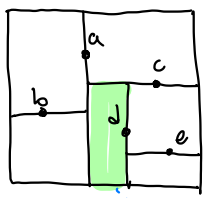
Tree height

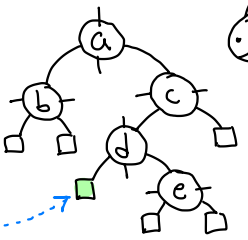
Run time: $O(h)$

## Can we balance the tree?

- Rotation does not make sense





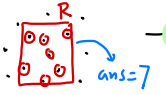




## Kd-Trees:

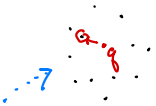
- Partition trees
- Orthogonal split → vert / horiz
- Alternate cutting dimension x, y, x, y, ...
- Cells are axis-aligned rectangles (AABB)


## 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







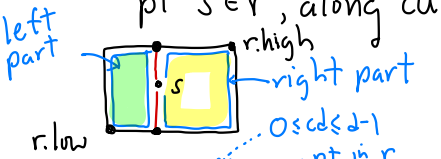

This Lecture: $O(\sqrt{n})$ time alg for orthog. range counting queries in $\mathbb{R}^2$

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

## Kd-Tree Queries I

## Rectangle methods for kd-cells:

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

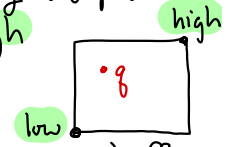



**r.leftPart(cd, s)**
→ returns rect with low = r.low & high = r.high but high[cd] ← s[cd]

**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



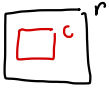

- Contains pt $q \in \mathbb{R}^d$ iff $low_i \le q_i \le high_i$ $1 \le i \le d$

## Useful methods:

Let r, c - Rectangle
q - Point

**r.contains(q)**


**r.contains(c)**



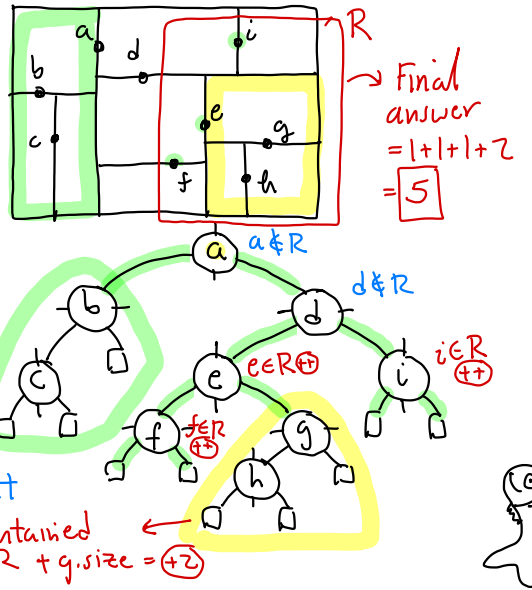

**r.isDisjointFrom(c)**

# 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.

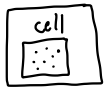
```
class Rectangle {
  private Point low, high
  public Rect(Point l, Point h)
    "    boolean contains(Point q)
    "    boolean contains(Rect c)
    "    Rect leftPart(int cd, Point s)
    "    Rect rightPart(" "  "  ")
}
```



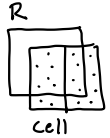

## Cases:

- 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 both children → Recurse on




```
int rangeCount(Rect R, KDNode p, Rect cell)
  if (p == null) return 0  // fell out of tree
  else if (R.isDisjointFrom(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 $n$

**Analysis:** How efficient is our algorithm?

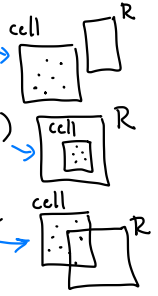
→ **Tricky to analyze**

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

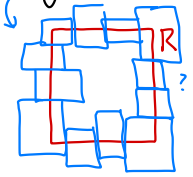
→ At some we don't recurse at all!

**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)

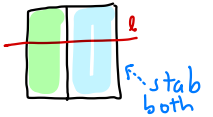



Simpler: Extend R's sides to 4 lines & analyze each one.

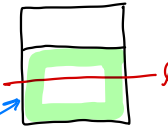
**Lemma:** Given a kd-tree (as in Thm above) and horiz. or vert. line $\ell$, at most $O(\sqrt{n})$ cells can be stabbed by $\ell$

**Proof:** w.l.o.g. $\ell$ is horiz.

**Cases:** p splits vertically




p splits horizontally




**Solving the Recurrence:**

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

**Master Thm:**

$$T(n) = aT\left(\frac{n}{b}\right) + n^d \quad * \; d < \log_b a$$

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

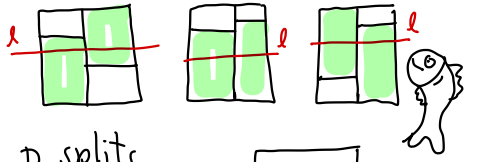
For us: $a = 2$, $b = 4$, $d = 0$ ⇒ $T(n) = n^{\log_4 2} = n^{1/2} = \sqrt{n}$

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

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

2 cells stabbed; Recurse on 2 grandchildren; Each has $n/4$ pts

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



# Hashing I

**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

**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? $O(1)$?

**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
- $h_{a,b}(x) = ((ax+b) \bmod p) \bmod m$

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

**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: $h$: 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

table: 


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

Assume keys can be interpreted as ints

# Hashing II

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

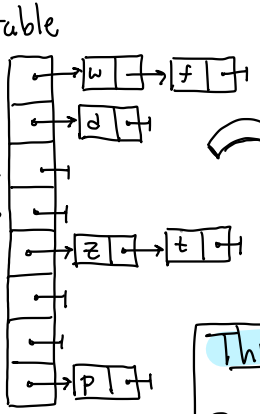
simple/slow → complex/fast

## Separate Chaining:
table[i] is head of linked list of keys that hash to i.

### Example:

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

m = 8




### Analysis:
Recall load factor $\lambda = n/m$
n = # of keys
m = table size

## Collision Resolution:
If there were no collisions hashing would be trivial!

insert(x,v) → table[h(x)] = v
find(x) → return table[h(x)]
delete(x) → table[h(x)] = null

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

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

**Proof:** On avg. each list has $n/m = \lambda$
success: 1 for head + half the list
unsuccess: 1 " " + all the list

## If $\lambda < \lambda_{min}$ or $\lambda > \lambda_{max}$? Rehash!
- Alloc. new table size = $n/\lambda_0$
- Compute new hash fn h
- Copy each x,v from old to new using h
- Delete old table

Token-based — See latex notes!

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

## How to control $\lambda$?
- Rehashing: If table is too dense / too sparse, realloc. to new table of ideal size

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

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

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

## Open Addressing:

- Special entry ("empty") means this slot is unoccupied
- Assume $\lambda \leq 1$
- To insert key:
  check: $h(x)$ if not empty try
  $h(x)+i_1$
  $h(x)+i_2$
  ⋮

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

- What's the best probe sequence?

## Collision Resolution: (cont.)

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

→ Open Addressing

Hashing III

## Analysis: Improves secondary clustering

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

**Thm:** If quad. probing used & $m$ is prime, the the first $\lceil m/2 \rceil$ probe locations are distinct.

**Pf:** See latex notes.

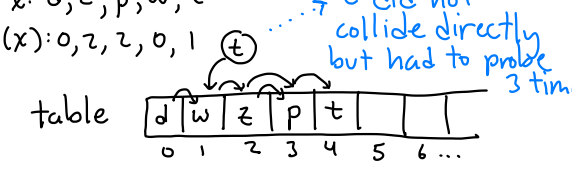
## Linear Probing:

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


until finding first available

### Simple, but is it good?

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




## Analysis:

Let $S_{LP}$ = expected time for successful search

$U_{LP}$ = " " unsuccessful "

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

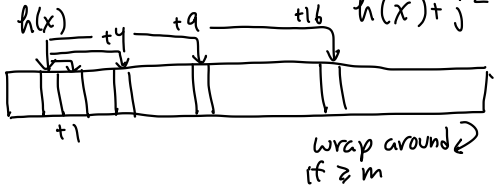
**Obs:** As $\lambda \to 1$, times increase rapidly

## Clustering

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

### Quadratic Probing:

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



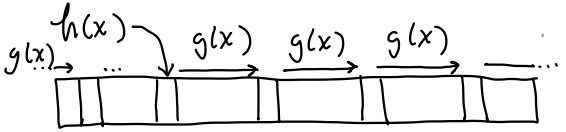
# Hashing IV

## Double Hashing:

(Best of the open-addressing methods)

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

$h(x) + \{0, g(x), 2\cdot g(x), 3\cdot g(x) \ldots\}$ [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 $\lambda = n/m$

## Recap:

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

**Open Addressing:**

- Linear probing: } clustering
- Quadratic probing: }

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

→ Proof is nontrivial (skip)

| $\lambda$: | 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!

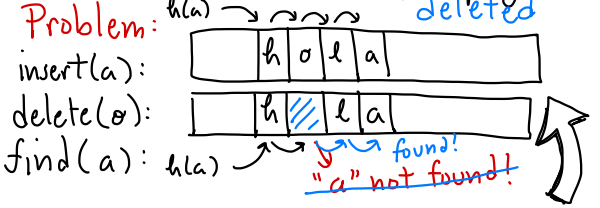
## Delete(x): Apply find(x)

→ Not found ⇒ error

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

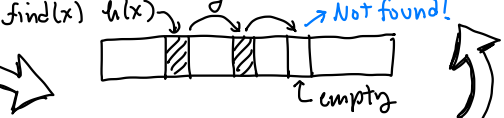
Is this right??

Problem:




## Find(x): Visit entries on probe sequence until:

- found x ⇒ return v
- hit empty ⇒ return null




## 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!)

## Scapegoat Trees:

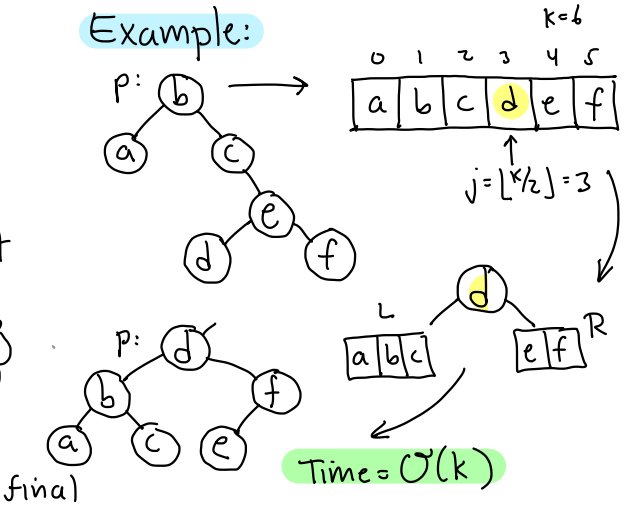
- Arne Anderson (1989)
- Galperin + Rivest (1993) rediscovered/extended
- **Amortized analysis**
  - $O(\log n)$ for dictionary ops amortized (guaranteed for find)
  - Just let things happen
  - If subtree unbalanced
    - rebuild it

## Recap:

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

Scapegoat Trees I

## Example:




## Overview:

**Insert:**

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

**Find:** Same as std BST

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

**Delete:**

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

**How?** Maintain $n, m \leftarrow 0$

Insert: $n{+}{+}, m{+}{+}$

Delete: $n{-}{-}$ ⋯→ If $m > 2n$ rebuild

## How to rebuild?

**rebuild(p):**

- inorder traverse p's subtree → array A[ ]
- buildSubtree(A)

**buildSubtree**(A[0..k-1]):

- if k=0 return null
- $j \leftarrow \lfloor k/2 \rfloor$; $x \leftarrow A[j]$ median
- $L \leftarrow$ buildSubtree(A[0..j-1])
- $R \leftarrow$ buildSubtree(A[j+1.. k-1])
- return Node(x, L, R)

# Scapegoat Trees II

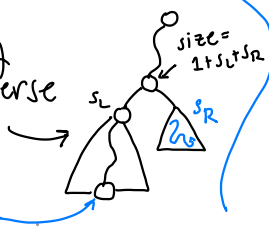
## Insert:

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

scapegoat

## How to compute size(p)?

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

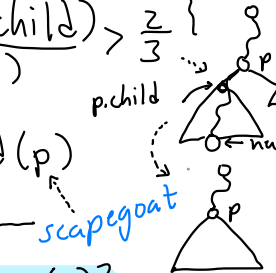



## Details of Operations:

**Init:** n ← m ← 0  root ← null

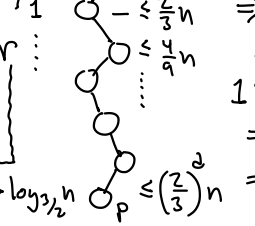
**Delete:**
- Same as std BST
- n--
- if m > 2n,
  rebuild(root)

Time: O(n)



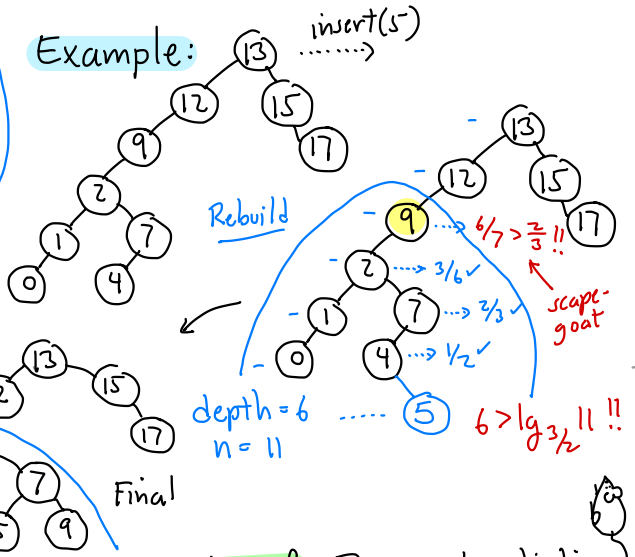

## Must there be a scapegoat? yes!

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




⇒ Since p has 1 node:
$1 \le \text{size}(p) \le (\frac{2}{3})^d n$
$\Rightarrow (\frac{3}{2})^d \le n$
$\Rightarrow d \le \log_{3/2} n$ □

## Example:




## Proof: By contradiction

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


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

**Proof:** (Sketch)

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

**Delete:** In order to induce a rebuild, number of deletes $\sim$ number of nodes in tree

→ Amortize rebuild time against delete ops

**Insert:** Based on potential argument

→ It takes $\sim k$ ops to cause a subtree to size $k$ to be unbalanced.

→ Charge rebuild time to these operations

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

→ Multi-Layer DataStructures

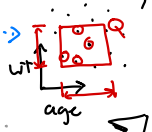
## Recap:

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

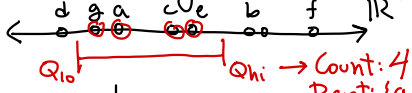

- kd-Tree: Counting: $O(\sqrt{n})$ time
  Report: $O(k + \sqrt{n})$ time

No. of pts reported

Range Trees I


## 1-Dim Range Tree:




Approach:

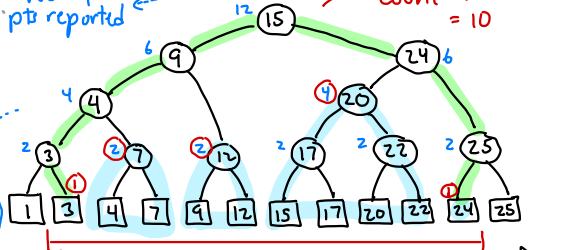
- Balanced BST (eg. AVL, RB, ...)
- Assume extended tree
- Each node p stores no. of entrie in subtree: p.size

## 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 $O(k + \log n)$)

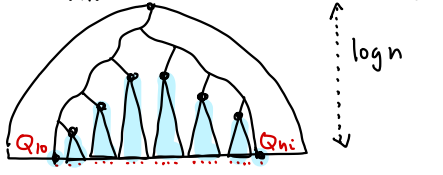
No. of pts reported

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

## Recursive helper:

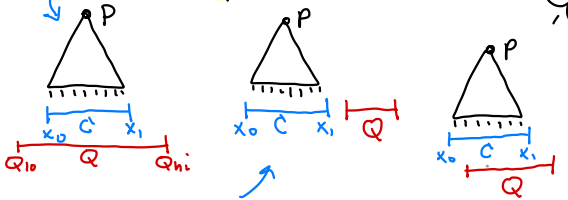
int range1Dx(Node p, Intv Q=[$Q_{lo}$,$Q_{hi}$], Intv C=[$x_0$,$x_1$])

initial call: range1Dx(root, Q, $C_0$)

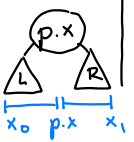
## Cases:

**p is external:**
- if p.pt.x ∈ Q → 1 else → 0

**p is internal:**
- C ⊆ Q ⇒ all of p's pts lie within query
  → return p.size

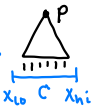



- C is disjoint from Q ⇒ none of p's pts lie in Q
  → return 0
- Else **partial overlap**
  → Recurse on p's children + trim the cell



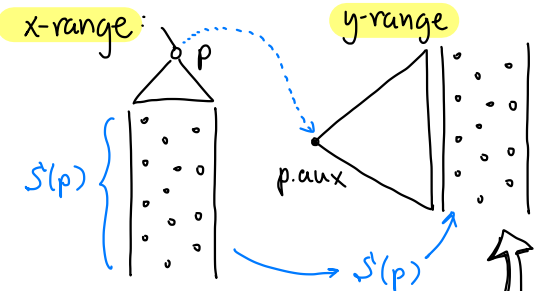

## More details:

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




```
int range1Dx(Node p,
    Intv Q, Intv C=[x0,x1]) {
  if(p is external)        → 1
    return p.pt.x ∈ Q      → 0
  else if (C ⊆ Q) return p.size
  else if (Q+C disjoint) return 0
  else return:
    range1Dx(p.left, Q, [x0, p.x])
  + range1Dx(p.right, Q, [p.x, x1])
```

## x-range: y-range




## 2-D Range Searching:

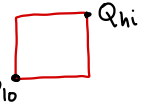
- "Layer" a range tree for x with range tree for y
- For each node p ∈ 1D-x tree, let S(p) = set of pts in p's subtree
- Def: **p.aux**: A 1D-y tree for S(p)

## 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.

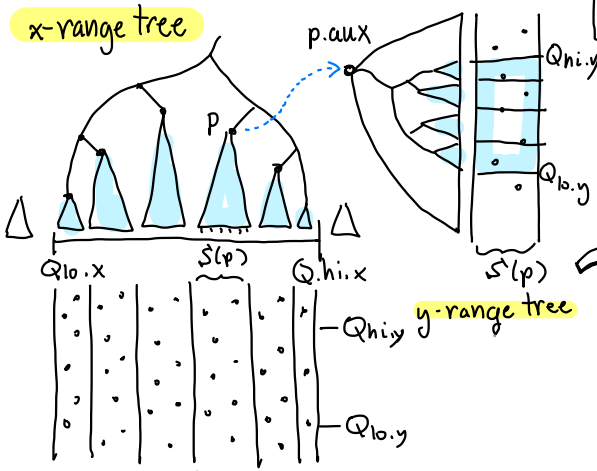
**Thm:** Given 1-D range tree... can answer range queries in time $O(\log n)$ ....→ (+k to report)

## 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 $y \to$ p.aux
- Final structure is union of x-tree + (n-1) y-trees

Range Trees III

## 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:

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

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

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

**Analysis:**

```
int range2D(Node p, Rect Q, Intv C=[x0,x1]){
  if (p is external) return p.pt ∈ Q? 1 : 0
  else if (Q.x contains C){   // C ⊆ Q's x-projection
    [y0,y1] = [-∞,+∞]         // init y-cell
    return range1Dy(p.aux, Q, [y0,y1])
  }else if (Q.x is disjoint of C) return 0
  else                        // partial x-overlap
    return range2D(p.left, Q, [x0, p.x])
         + range2D(p.right, Q, [p.x, x1])
}
```

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

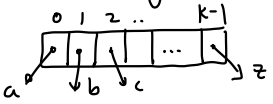
Invoked $O(\log n)$ times - once for each ancestor of max subtree

# Tries and Digital Search Trees I

## Tries: History

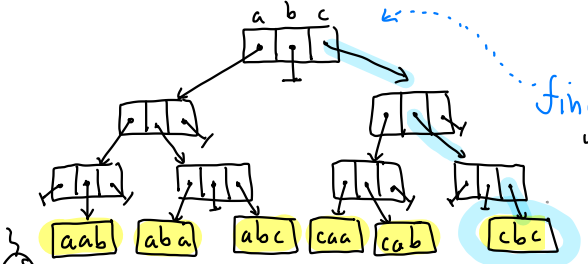
- de la Briandais (1959)
- Fredkin - "trie" from "retrieval"
- Pronounced like "try"

## Node: Multiway of order k



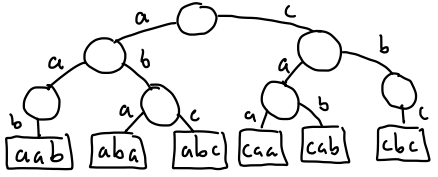

## Example: $\Sigma = \{a=0, b=1, c=2\}$

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




find("cbc")

## Same structure/Alt. Drawing




## Digital Search:

- Keys are strings over some alphabet $\Sigma$
- E.g. $\Sigma = \{a, b, c, \ldots\}$
  $\Sigma = \{0, 1\}$ 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:

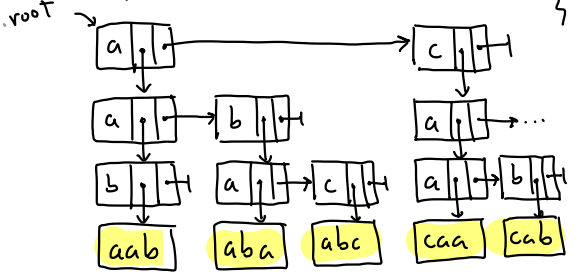
- No. of nodes ~ total no. of chars in all strings
- Space ~ k·(no. of nodes)

Large!

## Analysis:

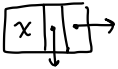
- Space: Smaller by factor k
- Search Time: Larger by factor of k

## Example:




## How to save space?

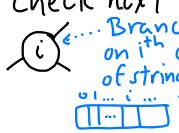
### de la Briandais trees:

- Store 1 char. per node
- 
  ≠ x ⇒ try next char in $\Sigma$
  == x ⇒ advance to next character of search string
- First-child/next-sibling

# Tries and Digital Search Trees II

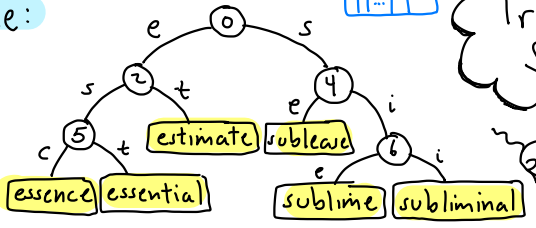
## Patricia Tries:

- 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)

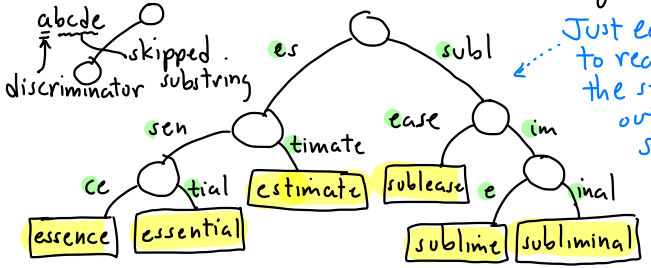



Example:

essence
essential
estimate
sublease
sublime
subliminal



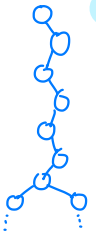

## Same data structure - Drawn differently




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 |

| | ID |
|---|---|
| $S_5$: ajam... | aj |
| $S_4$: pajam... | paj |
| $S_3$: apaja... | ap |
| $S_2$: mapaj... | map |
| $S_1$: amapaj... | amap |
| $S_0$: pamapa... | pam |

Example: $S$ = pamapajama$ (0 1 2 3 4 5 6 7 8 9 10)

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

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

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

## 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)

## Suffix Trees:

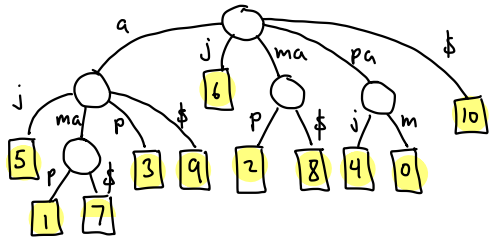
- Given single large text $S$
- Substring queries: "How many occurrences of "tree" in CMSC 420 notes"

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

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

# Tries and Digital Search Trees III

**Example:** $S$ = pamapajama$ (positions 0 1 2 3 4 5 6 7 8 9 10)




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

## Substring Queries:

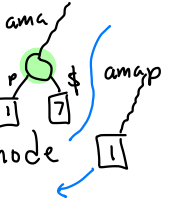
How many occurrences of t in text?

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

**Example:**

Search("ama") → End at intern node → Report: 2 occ's.

Search("amapaj") → End at extern node → Go to $S_1$ + verify




## 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, $O(n \cdot k)$ total space ($k = |\Sigma| = O(1)$)
- **Search time:** ~ to length of target string
- **Construction time:** $O(n \cdot k)$ [nontrivial]

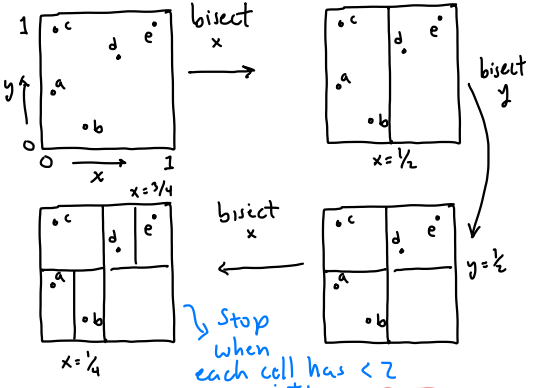
## PR k-d tree:

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

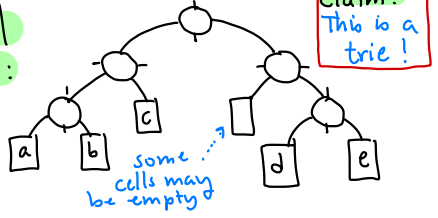
## Geometric Applications:

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

Assume points lie in unit square




**Final tree:**




**Claim:** This is a trie!

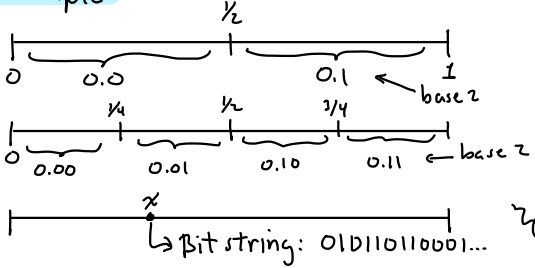
## Binary Encoding:

- 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:

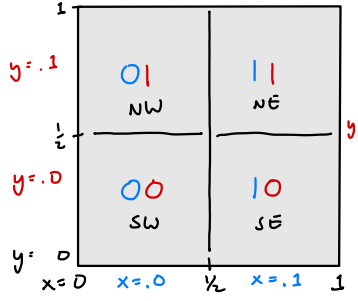
$x = 0.a_1a_2a_3\ldots \quad a_i \in \{0,1\}$

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

## Example:




## How do we extend to 2-D?

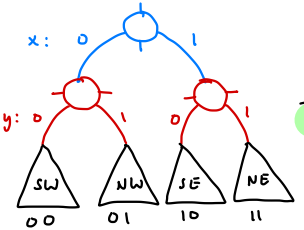



## PR kd-Tree ≡ Trie ??

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

Tries and Digital Search Trees IV

## PR 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

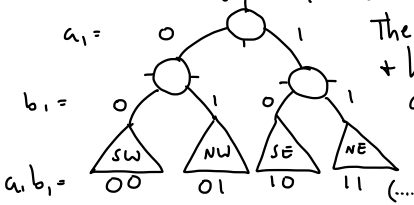
## Further Remarks:

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

$x = 0.a_1a_2\ldots$
$y = 0.b_1b_2\ldots$
$z = 0.c_1c_2\ldots$
$\Big\} \phi = a_1b_1c_1a_2b_2c_2\ldots$

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




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