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

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

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

**Common:**
- $O(1)$: constant time
  - [Hash map]
- $O(\log n)$: log time (very good!)
  - [Binary search]
- $O(n^p)$: ($p = \text{constant}$) Poly. time
  - e.g. $O(n^2)$

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

**Asymptotic Analysis:**
- Run time as a function of $n \gets \text{no. of items}$
- Worst-case, average-case, randomized
- Amortized: Average over a series of ops.

---

**Data structures are FUNDAMENTAL!**

- All fields of CS involve storing, retrieving and processing data
- Information retrieval
- Geographic Inf. Systems
- Machine Learning
- Text/String processing
- Computer graphics
  - ...

**Basic elements in study of data structures**

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

---

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

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

$T(n) = O(n^2)$
Linear List ADT:
Stores a sequence of elements \( \langle a_1, a_2, ..., a_n \rangle \). 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 \( i^{th} \) element (moving others back)
- delete(i) - deletes \( i^{th} \) item (moving others up)
- length() - returns num. of items

Implementations:
- Sequential: Store items in an array
  \[
  a_1 \ a_2 \ a_3 \ ... \ a_n
  \]
- Linked allocation: linked list
  \[
  \text{Singly: } \quad \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{null}
  \]
  \[
  \text{Doubly: } \quad \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{tail}
  \]
- Performance varies with implementation

Abstract Data Type (ADT):
- Abstracts the functional elements of a data structure (math) from its implementation (algorithm/programming)
- Doubles 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

Basic Data Structures I
- ADTs
- Lists, Stacks, Queues
- Sequential Allocation
  \[
  \begin{align*}
  \text{Stack: } & \quad \text{All access from one side} \\
  & \quad \text{push} + \text{pop} \\
  \text{Queue: } & \quad \text{FIFO list: enqueue inserts at tail and dequeue deletes from head}
  \end{align*}
  \]
Cost model (Actual cost)
- Cheap: No reallocation → 1 unit
- Expensive: Array of size \( n \) \( \geq 2n+1 \)
is reallocated to size \( 2n \)

Dynamic (Sequential) Allocation
- When we overflow, double

Eg. 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 \( \leq 5m \)]

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
- 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 4\( n \), but we have enough saved!
### Fixed Increment
Increase by a fixed constant
\[ n \rightarrow n + 100 \]

### Fixed factor
Increase by a fixed constant factor (not nec. 2)
\[ n \rightarrow 5 \cdot n \]

### Squaring
Square the size (or some other power)
\[ n \rightarrow n^2 \text{ or } n \rightarrow n^{1.5} \]

***Which of these provide \( O(1) \) amortized cost per operation?***

Leave as exercise
(Spoiler alert!)
- **Fixed increment** → no
- **Fixed factor** → yes
- **Squaring** → ?? (depends on cost model)

### Dynamic Stack
- Showed doubling ⇒ Amortized \( O(1) \)
- Other strategies?

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

---

### Node

### Idea: Store only non-zero entries linked by row and column

---

### Multilists: Lists of lists

---

### Sparse Matrices:
An \( nxm \) matrix has \( n \cdot m \) entries and takes (naively) \( O(n \cdot m) \) space

Sparse matrix: Most entries are zero
Tree (or "Free Tree")
- undirected
- connected
- acyclic graph

Rooted tree: A free tree with root node

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

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

"Family" Relations
- grandparent
- parent
- child
- sibling
- grandchild

Leaf: no children
Representing rooted trees: Each node stores a (linked) list of its children.

Wasted space?

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

E.g., \( n=4 \) 
nulls: \( 6 \)

In Java:

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

Node structure:

- `data`
- `firstChild`
- `nextSibling`

Trees Representation + Binary Trees (I)

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

Full: Every non-leaf node has 2 children.

Called the Binary representation.
traverse (BTreeNode v) {
    if (v == null) return;
    visit/process v ← Preorder
    traverse (v. left) ← Inorder
    visit/process v ← Postorder
    traverse (v. right)
    visit/process v
}

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

Binary Tree Traversals (can be generalized)

- process/visit v
- traverse T_L (recursive)
- traverse T_R

Complete Binary Tree: All levels full (except last)

Challenge: Non-recursive traversals

Preorder: 1 * + a b c - d e
Inorder: a b * c / d - e
Postorder: a b c * d e - 1

Binary Trees: Traversals, Extension, and More

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

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

Observation: Every extended binary tree is full

Observation: Every extended binary tree is full

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

Eg. Inorder Threads:
Null left → inorder predecessor
Null right → "successor
Examples:
- Given prime $p$, $a \equiv b \mod p$
  
  \[ \text{Eg. } p=5; \text{Partition: } \{0,5,10,\ldots\},\{1,6,11,\ldots\},\{2,7,12,\ldots\},\ldots \] 

- Given graph $G$, vertices $u, v$, $u \equiv v$ if in same connected component

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

Equivalence Relation:
Binary relation over set $S$ such that \( \forall a, b, c \in S: \)
  - reflexive: \( a \equiv a \)
  - symmetric: \( a \equiv b \Rightarrow b \equiv a \)
  - transitive: \( a \equiv b \land b \equiv c \Rightarrow a \equiv c \)

An equivalence relation defines a partition over $S$.

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

\[ \text{Set Simple-Find(\text{Element } x) } \{ \]
  \[ \text{while} \ \text{parent}[x] \neq \text{null} \]
  \[ x \leftarrow \text{parent}[x] \]
  \[ \text{return } x \]

Set Identifiers are indices of root nodes

\[ \text{Eg. } \text{Find}(7) = 3 \]
\[ \text{Find}(10) = 3 \]
\[ \text{Find}(5) = 2 \]

Two items in same set iff \( \text{Find}(x) = \text{Find}(y) \)

Inverted-Tree Approach:
- Store elements of each set in tree with links to parent
- Root node is set identifier

\[ \text{Eg. } \{1,3,7,10\}, \{1\}, \{2,5,6,8,11\}, \{4,9\} \]

Disjoint Set Union-Find I

Array-Based Implementation:
Assume: \( S = \{1,2,\ldots,n\} \)

\[ \text{parent}[1..n], \text{where } \text{parent}[i] \text{ is } i \text{ if root} \]

\[ \text{parent index or } 0 = \text{null if root} \]

\[ 1 2 3 4 5 6 7 8 9 10 11 12 \]

\[ 3 0 0 9 11 2 3 2 0 1 2 0 \]
Set Union (Set s, Set t) 
\[
\begin{align*}
&\text{if (rank}[s]\rangle \text{rank}[t]) \\
&\quad \text{swap } s \leftrightarrow t \\
&\quad \text{parent}[s] \leftarrow t \\
&\quad \text{rank}[t] \leftarrow \max\left(\text{rank}[t], 1 + \text{rank}[s]\right) \\
&\quad \text{return } t
\end{align*}
\]

Recall: These are just array indices of roots

How to Union?
- Just link one tree under the other
- How to maintain low heights?
- Rank: Based on height of tree. Link lower rank as child

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

Proof: By induction on num. of unions

Basis: Single node. \( h = 0 \), \( 2^0 = 1 \) nodes

Step: Consider the last of series of unions. Let \( T' + T'' \) be trees to merge. Heights: \( h' + h'' \)
Sizes: \( n' + n'' \)

By induction: \( n' \geq 2^h \), \( n'' \geq 2^h \)

Cases: \( h' = h'' \)
- Union: \( O(1) \) - constant time
- Find: \( O(\text{tree height}) \)

Running Time?
- Init: \( O(n) \) - set a parents to null + ranks to 0
- Union: \( O(1) \) - constant time

What is worst case?
We’ll show tree height = \( O(\log n) \)
\Rightarrow \text{Find takes } O(\log n) \text{ time}

Example:
\[
\begin{align*}
\{1,3,7,10\} & \quad \{13\} \quad \{2,5,6,11\} \quad \{4,9\} \\
\begin{array}{c}
6 & \quad 7 & \quad 3 \\
\end{array} & \quad \circ & \quad \circ & \quad \circ & \quad \circ \\
10 & \quad 12 & \quad 8 & \quad 11 & \quad 5
\end{align*}
\]

Union (9, 12)
- [r2 has lower rank]
- rank[9] = \max(\text{rank}[9], 1 + \text{rank}[12])
  = \max(1, 1)
  = 1

Union (2, 3)
- [Both have same rank]
- rank[3] = \max(\text{rank}[3], 1 + \text{rank}[2])
  = \max(2, 3)
  = 3

Final tree height: \( h = h' + 1 = h'' + 1 \)
Final size:
\[
\begin{align*}
n & = n' + n'' \\
& \geq 2^h + 2^h - 2 - 2^{h-1} \\
& = 2^h
\end{align*}
\]

Case 2: \( h' < h'' \)
Final height: \( h = h'' \)
Final size:
\[
\begin{align*}
n & = n' + n'' \\
& \geq 2^h + 2^h \geq 2^h = 2^h
\end{align*}
\]

Case 3: \( h' > h'' \) (symmetrical)
\[
\begin{align*}
\text{Init: } & \quad \text{All ranks } \leftarrow 0 \\
\text{Union: } & \quad \text{O(n) - set a parents to null + ranks to 0} \\
\text{Find: } & \quad \text{O(1) - constant time} \\
\text{ Worst case: } & \quad \text{O(\log n) - tree height} \\
\text{What is worst case? } & \quad \text{We’ll show } \text{tree height } = \text{O(\log n)} \\
\text{Find takes } & \quad \text{O(\log n) time} \\
\end{align*}
\]
Path Compression:
- Whenever we perform find, shortcut the links so they point directly to root.
- This does not increase running by more than constant, but can speed up later finds.

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

⇒ Amortized time (average per op) is $O(\log m)$.
- Not bad, but can we do better?

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

⇒ Amortized time = $O(\alpha(m,n))$

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

Example:

Example:

Digression: Ackerman’s Function
(1926)

Does this little trick improve running times?

- Worst case - No find may take $O(\log n)$ time.
- Amortized - Yes! Huge improvement! (But hard to prove.)

Digression: Ackerman’s Function
(1926)

$$\alpha(m,n) = \min \{ i \geq 1 | A(i, \left\lceil \frac{m}{n} \right\rceil) > \log m \}$$

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

From super big to super small

Inverse of Ackerman

Disjoint Set Union-Find

(A.phas.)(a.phas.)(a.phas.)(a.phas.)

More than atoms in universe

More than atoms in universe

More than atoms in universe
Naive Solution:
- Store items in linear list
- Order?
  - Insert order - fast insert/slow extract
  - Priority order - fast extract/slow insert

Heap: Tree-based structure
(min) heap order: for all nodes, parent’s key ≤ node’s key
[Reverse: max-heap order]

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

Binary Heap:
- Simple, elegant, efficient
- Old (1964)
- Basic: insert/extract O(log n)
  - Build: O(n)

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

Priority Queues + Heaps I

Priority Queues + Heaps II

Void insert(key x)
- n++; i ← sift-up(n,x)
  A[i] ← x
int sift-up(int i, key x)
- while (i > 1 && x < A[par(i)])
  - A[i] ← A[par(i)]
  - i ← par(i)
- return i

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

```
×    4
✓  3
✓  12
✓  22
✓  8
✓  13
✓  14
✓  9
✓  24
✓  21
```

**Binary Heap - Extract Min**

- Min key at root $\rightarrow$ save it
- Copy $A[n]$ to root ($A[i]$) + decrement $n$
- Sift the root key down
  - Find smaller of two children
  - If larger, swap with this child
- Return saved root key

**Priority Queues & Heaps II**

**Leftist Property:** Null path length

\[
npl(v) = \text{length of shortest path to null}
\]

\[
npl(v) = \begin{cases} 
-1 & \text{if } v = \text{null} \\
1 + \min(npl(v.\text{left}), npl(v.\text{right})) & \text{ow.}
\end{cases}
\]

**Def:** Leftist Heap is binary tree where:
- Keys are heap ordered
- $\forall$ nodes $v$, $npl(v.\text{left}) \geq npl(v.\text{right})$

**Examples**

```
6
3
10
```

- Not Leftist!
- Leftist

```
6
3
10
```

```
12
7
```

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

**Analysis:** Both insert + extract-min take time proportional to tree height

Tree is complete $\Rightarrow O(\log n)$ time
Class structure:

Leftist Heap \(<\text{Key}>\) 

private class LHNode {
    [key x, LHNode left, right]
    int npl
}

private LHNode root

public LeftistHeap() {root = null}

    "constructor"
    public function

    ... (other private/protected utilities)

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

Merge helper: 2 phases

1. Merge right paths by order of keys + update npl's
2. Check leftist property + swap

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

Proof: (Sketch - see latex notes)

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

Insert + Extract-min

Exercises

Private class LHNode

has \( n \geq E-1 \) nodes

Key x

class Proof:

Sketch - see latex notes

LHNodemerge.LHNode u, LHNode v)

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

return u

Priority Queues + Heaps III

Phase 1

Phase 2

Final tree!
Dictionary:
- `insert (Key x, Value v)`
  - insert `(x, v)` in dict. (No duplicates)
- `delete (Key x)`
  - delete `x` from dict. (Error if `x` not there)
- `find (Key x)`
  - returns a reference to associated value `v`, or null if not there.

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

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

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

Efficiency: Depends on trees' height
- Balanced: $O(\log n)$
- Unbalanced: $O(n)$

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

Binary Search Trees
- Basic definitions
- Finding keys

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!

Example:
```
root
    / \
   7   12
  / \   |
 4   8 16
 / \   |
1   7 11
```

Value:
```java
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"

Replacement Node?

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)?

3. x has two children

Find replacement node
p.right-insert (x, p.right), copy to x, and then delete x

Binary Search Trees II
- insertion
- deletion

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

3 cases:
1. x is a leaf
2. x has single child
Find Replacement Node

```java
BSTNode findReplacement(BSTNode p) {
    BSTNode r = p; // Copy r's contents to p
    while (r.left != null) {
        r = r.left;
    }
    return r;
}
```

**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 Ens
- 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

**Example:**
- `del(3)`
- `del(4)`
Java implementation (see notes for details)

```java
public class BsTree<Key extends Comparable, Value> {

    class Node {
        Key key;
        Value value;
        Node left, right;
        ...
    }

    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: \( \text{bal}(v) = \text{hgt}(v.\text{right}) - \text{hgt}(v.\text{left}) \)

AVL Height Balance
- For each node \( v \), the heights of its subtrees differ by \( \leq 1 \).

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

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 \( h \))
- \( h = 0 \): \( n(0) = 1 = F_3 - 1 \)
- \( h = 1 \): \( n(1) = 2 = F_4 - 1 \)
- \( h \geq 2 \):
  \[ n(h) = 1 + n(h-1) + n(h-2) \]
  \[ = 1 + (F_{h+2} - 1) + (F_{h+1} - 1) \]
  \[ = (F_{h+2} + F_{h+1}) - 1 = F_{h+3} - 1 \]

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

Proof: Fact: \( f_h \approx \phi^h / \sqrt{5} \) where \( \phi = (1 + \sqrt{5})/2 \) "Golden ratio".

\[ n \geq \phi^h \Rightarrow h \leq \log_{\phi} n + c \]

\[ \Rightarrow h \leq \log_{\phi} n / \log_{\phi} \phi \]

\[ = O(\log n) \]
AVL Tree:
AVL Node: Same as BSTNode (from Lect 4) but add: int height

Utilities:
int height (AVLNode p) return \{ p == null \rightarrow -1, o.w. \rightarrow p.height \}

void updateheight(AVLNode p) p.height = 1 + \max \{ \text{height}(p.left), \text{height}(p.right) \}

int balance factor (AVLNode p) return \text{height}(p.right) - \text{height}(p.left)

AVL Node rebalance (AVL Node p)
if (p == null) return p
if (balanceFactor(p) < -1)
  if (height(p.left.left) \geq \text{height}(p.left.right))
    p = rotateRight(p)
  else p = rotateLeftRight(p)
else if (balanceFactor(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)

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

How to rebalance? Bal = -2

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

AVL Trees III - Deletion - Examples
Node types:
- 2-Node: 1 key, 2 children
- 3-Node: 2 keys, 3 children

Recap:
- AVL: Height balanced
- Binary
- 2-3 tree: Height exact
- Variable width

AVL: Height balanced
| 1 + 3 = 2 + 2 |

Adoption (Key Rotation):
| 1 + 2/2 + 1 → 3 |

Merge:
- steal b from parent

Split:
- insert in parent

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.

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

Example:
- 2-3 tree of height 2

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

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

2-3 Trees
Insertion example:

\[
\begin{align*}
\text{Dictionary operations:} \\
\text{Find} & \text{ - straightforward} \\
\text{Insert} & \text{ - find leaf node} \\
\text{where key “belongs”} & \text{ + add it (may split)} \\
\text{Delete} & \text{ - find/replacement/merge or adopt}
\end{align*}
\]

Implementation?

```java
class TwoThreeNode {
  int children [3];
  key key [2];
}
```

**2-3 Trees**

Delete Example:

\[
\begin{align*}
\text{Deletion remedy:} \\
\text{Have a 3-node neighboring sibling} & \text{ adopt} \\
\text{O.w.: Merge with either sibling} & \text{ + steal key from parent}
\end{align*}
\]

Example (continued)
Encoding 3-node as binary tree node

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

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

Lemma: A red-black tree with n keys has height \(O(\log n)\)
Proof: It's at most twice that of a 2-3 tree.

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

AA-Trees: Simpler to code
- No null pointers: Create a sentinel node, nil, and all nulls point to it → nil:
- No colors: Each node stores level number. Red child is at same level as parent. 
  \(q\) is red \(\iff q\) level == p.level

What we need are stricter rules!

AA-tree:
Arne Anderson 1993
New rule:
6. Each red node can arise only as right child (of a black node)

Nope! Alternatives that satisfy rules:

A "left-skewed" encoding corresponds to 2-3-4 trees
Restructuring Ops:
- **Skew**: Restore right skew
  - If black node has red left child, rotate
    - How to test? \( p . \text{left} . \text{level} = p . \text{level} \)

**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.
- How to test? \( p . \text{level} = p . \text{right} . \text{level} = p . \text{right} . \text{right} . \text{level} \)

Example:
- **2-3 Tree**:
- **AA Tree**:

Red-Black + AA Trees II

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

```
AA Node \text{skew}(AA Node p)
if (p == nil) return p
if (p . left . level == p . level) right rotate p
AA Node q = p . right
p . left = q . right; q . right = p
q . level += 1 \text{ move } q \text{ up a level}
return q \text{ all okay}
else return p \text{ everything's fine}
```
Example: AANode insert(Key x, Value v, AANode p)
if (p == nil)
p = new AANode(x, v, 1, nil, nil)
else if (x < p.key) ... insert on left
else if (x > p.key) ... insert on right
else Duplicate Key:
   return split(skew(p))

Red-Black and AA Trees II

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

fix AfterDelete (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 fix AfterDelete (p)
Geometric Search:
- Nearest neighbors
- Range searching
- Point Location
- Intersection Search

Sofar: 1-dimensional keys
- Multi-dimensional data

Partition Trees:
- Tree structure based on hierarchical space partition
- Each node is associated with a region - cell
- Each internal node stores a splitter - subdivides the cell

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

Applications:
- Spatial databases & maps
- Robotics & Auton. Systems
- Vision/Graphics/Games
- Machine Learning

Quadtree & kd-Trees I

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

Class Point{
  float[] coord // coords
  Point(int d)
  \ldots coord = new float[d]
  int getDim() \rightarrow coord.length
  float get(int i) \rightarrow coord[i]
  \ldots others: equality, distance, toString...
Point Quadtree:
- Each internal node stores a point
- Cell is split by horiz. + vertic. lines through point

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

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

Quadtree: (abstractly)
- Partition trees
- Cell: Axis-parallel rectangle
- Splitter: Subdivides cell into four (generally 2^n) subcells

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)

Each external node corresponds to cell of final subdivision

Quadtree - Analysis
- Numerous variants!
- Popular in 2-d apps (in 3-d, octtrees)
- Don't scale to high dim
- What to do for higher dims?

History: Bentley 1975
- Called it 2-d tree (R^2)
- 3-d tree (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 ow.)
- Cell: Still AABB
  left: left/below
  right: right/above

Quadtree & kd-Trees II

IR' AABB
Axis-aligned bounding box
In short kd-tree Can I dim

Cell is split by horiz. ir vertu
[AABB - "Axis-aligned bounding box"
- In short kd-tree Can I dim

kd-Tree: Binary variant of quadtree
- Splitter: Horiz. or vertic. line in 2-d (orthogonal plane ow.)
- Cell: Still AABB
  left: left/below
  right: right/above
Example:

Kd-Tree Node:

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

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)

Quad-trees &
Kd-Trees III

How do we choose cutting dim?
- Standard Kd-tree: cycle through them (e.g. d = 3: 1, 2, 3, 1, 2, 3...)
- Based on tree depth
- Optimized Kd-tree: (Bentley)
  - Based on widest dimension of pts in cell.

Value
```
find(Point q, KDNode p) {
    if (p == null) return null;
    if (q == p.point) return true;
    if (q[cutDim] < p.point[cutDim]) return find(q, p.left);
    if (q[cutDim] > p.point[cutDim]) return find(q, p.right);
}
```

Helper:
```
class KDNode {
    boolean onLeft(Point q) {
        return q[cutDim] < pt[cutDim];
    }
}
```
Kd-Tree Insertion:
(Similar to std. BSTs)
- Descend tree until
  - find pt → Error: duplicate
  - falling out → (Although we draw extended trees, let's assume standard trees)
- create new node
- set cutting dim

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.

Can we balance the tree?
- Rotation does not make sense!!

Example:
\begin{itemize}
  \item insert(3,4)
\end{itemize}

\begin{itemize}
  \item insert((3,4))
  \item insert((4,5))
  \item insert((4,1))
  \item insert((5,5))
\end{itemize}
**Kd-Trees:**
- **Partition trees**
- **Orthogonal split**
- **Alternate cutting dimension** \(x, y, x, y, \ldots\)
- 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 \(\mathbb{R}^d\): \(O(n^{1-\frac{1}{d}})\)

**Rectangle methods for kd-cells:**
- Split a cell \(r\) by a split pt \(s\) ∈\(r\), along cutdim \(c\)
- \(r\).leftPart\((cd, s)\)
  - returns rect with \(\text{low} = r\.low\)
  - \(\text{high} = r\.high\) but \(\text{high}[\text{cd}] \leftarrow s[\text{cd}]\)
- \(r\).rightPart\((cd, s)\)
  - \(\text{high} = r\.high + \text{low} = r\.low\) but \(\text{low}[\text{cd}] \leftarrow s[\text{cd}]\)

**Axis-Aligned Rect in \(\mathbb{R}^d\)**
- Defined by two pts: \(\text{low}, \text{high}\)
  - Contains pt \(q\) ∈\(\mathbb{R}^d\) iff \(\text{low}_i \leq q_i \leq \text{high}_i\)
- \(\text{r}.contains(q)\)
- \(\text{r}.contains(c)\)
- \(\text{r}.isDisjointFrom(c)\)

**Useful methods:**
- Let \(r, c\) - Rectangles
- \(q\) - Point
- \(\text{r}.contains(q)\)
- \(\text{r}.contains(c)\)
- \(\text{r}.isDisjointFrom(c)\)
Orthogonal Range Query

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

Cases:
- $p == null$: fell out of tree $\rightarrow 0$
- Query rect is disjoint from $p$'s cell $\rightarrow \text{return 0}$
- Query rect contains $p$'s cell $\rightarrow \text{return } p.size$
- Otherwise: Rect. + cell overlap $\rightarrow \text{Recurse on both children}$

Kd-Tree Queries

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

```
int rangeCount (Rect R, KDNode p, Rect cell) {
  if (p == null) return 0 // fell out of tree
  else if (R. is Disjoint From (cell)) return 0 // 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...)
    return ct;
  }
}
```
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.

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!

Solving the Recurrence:

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

**Master Thm**:

$$T(n) = aT\left(\frac{n}{d}\right) + n^d + d\log_b a$$

For us: $a=2$, $b=4$, $d=0$:

$$T(n) = n^\log_4 2 = n^{\frac{1}{2}} = \sqrt{n}$$

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

**Recurrence**: $T(n) = 2 + 2T\left(\frac{n}{4}\right)$

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

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

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

Cases: $p$ splits vertically

Cases: $p$ splits horizontally

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

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

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

**Delete:**
- Same as std. BST
  - if num. of deletes is large rel. to $n$ -
    - rebuild entire tree!

**Find:** Same as std BST
- Tree height $\leq \log_{3/2} n \approx 1.71\log n$

**Example:**
- $k=6$
  - $j=[\lfloor 4/2 \rfloor]=3$

**How to rebuild ($p$):**
- inorder traverse $p$'s subtree $\to$ array $A[]$
- $buildSubtree(A)$

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

- **Insert:**
  - \( n++ \); \( m++ \)
  - Same as std BST but keep track of inserted node's depth \( \rightarrow d \)
  - If \( d > \log_{3/2} m \) \{ /* rebuild event */ \}
- **Delete:**
  - Same as std BST
  - \( n-- \)
  - If \( m > 2n \), rebuild \((\text{root})\)

**Scapegoat Trees II**

**Proof:** By contradiction

- Suppose p's depth \( > \log_{3/2} n \)
- But \( \forall \) ancestors \( u \), \( \text{size}(u, \text{child}) \leq \frac{2}{3} \cdot \text{size}(u) \)
- Node p of depth \( > \log_{3/2} n \), then \( \exists \) ancestor of p that satisfies scapegoat condition

**Insert:***

\[ n++; m++ \]

- Same as std BST but keep track of inserted node's depth \( \rightarrow d \)
- If \( d > \log_{3/2} m \) \{ /* rebuild event */ \}
- Trace path back to root
- For each node p visited, \( \text{size}(p) = \text{no. of nodes in } p's \text{ sub tree} \)
- If \( \frac{\text{size}(p, \text{child})}{\text{size}(p)} > \frac{2}{3} \)
  - \( p \rightarrow \text{rebuilt} \)
  - Break

**How to compute size(p)?**

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

**Example:**

**Scapegoat condition:**

- \( n \leq (3/2)^d \leq n \)
- \( d > \log_{3/2} n \)

**Lemma:** Given a binary tree with \( n \) nodes, if \( \exists \) node p of depth \( > \log_{3/2} n \), then \( \exists \) ancestor of p that satisfies scapegoat condition
Theorem: Starting with an empty tree, any seq. of $k$ inserts + deletes takes total of $O(k \log k)$ time.

Corollary: Amortized time is $O(\log k)$

Proof: Token-based argument
   
   Overview:
   - We will assign tokens to nodes of tree
   - Add some tokens "on the side"
   - Will show:
     - Total tokens = $O(k \log k)$
     - Enough tokens to pay for all rebuildings

   Token assignment:
   - Whenever we insert/delete, add a token to each node visited in the search
   - During each deletion - add 1 token "on the side"
   - By height bound - $O(k \log k)$ token total

Amortized Analysis:
- Tree height is $O(\log n)$
  [since we rebuild whenever higher]
  \[ \Rightarrow \text{find is } O(\log n) \text{ even in worst case} \]
- But insert + delete can take up to $O(n)$ time (if entire tree is rebuilt)

Therefore - node $u$ has collected at least $\frac{1}{3} \text{size}(u)-1$ tokens

Since it takes $O(\text{size}(u))$ time to rebuild $u$, it follows that (up to adjusting constants) we have enough tokens to pay for rebuild.

The last time a subtree containing $u$ was rebuilt, it was perfect balanced
\[ \Rightarrow \text{size}(u \text{.left})-\text{size}(u \text{.right}) \leq 1 \] (at that time)
This implies that since last rebuild, we had at least $\frac{1}{3} \text{size}(u)-1$ inserts/deletes involving $u$.

This implies:
\[ \frac{1}{2} \text{size}(u \text{.left}) > \text{size}(u \text{.right}) \]
\[ \Rightarrow \text{size}(u \text{.left})-\text{size}(u \text{.right}) > \frac{1}{2} \text{size}(u \text{.left}) \]
\[ > \frac{1}{3} \text{size}(u) \]

Claim: There are always enough tokens to pay for rebuilding.

Proof:
If we call $\text{buildSubtree}(u)$, we know $u$ is a scapegoat. Assume w.l.o.g. that:
\[ \frac{\text{size}(u \text{.left})}{\text{size}(u)} > \frac{2}{3} \]
By def: $\text{size}(u) = \text{size}(u \text{.left}) + \text{size}(u \text{.right})$
Other/Better Criteria?
- Expected case: Some keys more popular than others
- Self-adjusting: Tree adapts as popularity changes

How to design/analyze?
- Splay Tree: A self-adjusting binary search tree
  - No rules! (yay anarchy!)
  - No balance factors
  - No limits on tree height
  - No color/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

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 color/levels/priorities

Tree's height has reduced by ~ half!

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

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

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

find("a")

ugh!

→ Tree restructures itself
ZigZig(p): [LL case]

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) zigzag(p)
    else /* p is LR or RL grand child */ zigzag(p)
}

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

Example:
splay(3)

Insert (x):
Node p ← splay(x)
if (p.key == x) Error!!
q ← new Node(x)
if (p.key < x)
p.left ← p
p.right ← p.right
else /* (symmetrical) */
  root ← q

Splay Trees II

find(x):
root ← splay(x)
if (root.key == x)
  return root.value
else return null
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 \( p_i \): \( \sum p_i = 1 \).

Dynamic Finger Theorem:

Keys: \( x_1, \ldots, x_n \). We perform accesses \( x_{i_1}, x_{i_2}, \ldots, x_{i_m} \).

Let \( \Delta_j = i_j - i_{j-1} \), distance between consecutive items.

Let \( D_j = i_j - p \), distance from \( i_j \) to \( p \).

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

Given a seq. of \( m \) ops. on splay tree with keys \( x_1, \ldots, x_n \), where \( x_i \) is accessed \( q_i \) times. Let \( p_i = q_i/m \). Then total time is
\[ O(m \sum p_i \log y_i) \]
Ideal Skip List:
- Organize list in levels
  - Level 0: Everything
    1: Every other
    2: Every fourth
  i: Every $2^i$
- 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:
```
class SkipNode
    Key key
    Value value

    SkipNode[] next
```

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--; \(\leftarrow\) drop down a level
\}
we are at base level

if (p.key == x) return p.value
else return null
```
**Thm:** A skip list with $n$ nodes has $O(\log n)$ levels in expectation.

**Proof:** Will show that probability of exceeding $c \cdot \log n$ is $\leq \frac{1}{n^c}$.

- $\Pr$ that any given node's
  level exceeds $l$ is $\frac{1}{2^l}$
  ($l$ consecutive heads)
- $\Pr$ that any of $n$ node's
  levels exceeds $l$ is $\leq \frac{n}{2^l}$
  (n trials with prob $\frac{1}{2^l}$)
- Let $l = c \cdot \log n$ ($c = \log_2 e$)
  Prob that max level exceeds $c \cdot \log n$ is:
  $\leq \frac{n}{2^l} = \frac{n}{2^{c \cdot \log n}}$
  $= n/(2^{\log_e n})^c$
  $= n/n^c = 1/n^{c-1}$

**Obs:** Prob. level exceeds $3 \cdot \log n$ is $\leq \frac{1}{n^2}$.
(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 $\frac{1}{2^i}$.
- Expected no. of nodes that contrib. to level $i = n/2^i = E(n_i) = \frac{n}{2^i}$

Total space (expected) is:

$E(\sum_{i=0}^{\max} E(n_i)) = \sum_{i=0}^{\max} \frac{n}{2^i}$

$= n \sum_{i=0}^{\max} \frac{1}{2^i} = 2n$

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

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

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

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

**Cases:**

**A:**

$E(i) = 1 + (\Pr(A))E(i) + (\Pr(\overline{A}))E(i-1)$

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

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

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

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

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

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

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

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

Example: find(25)

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 coin to determine height)
  - Link into each saved node

Example: Insert(24)

Analysis: All operations run in time $\sim \text{find} \Rightarrow O(\log n)$ expected
Note: Variation in running times due to randomness only - not sequenc. $\Rightarrow$ User cannot force poor performance.
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: \(O(\log n)\)
  \rightarrow\) Worst-case, amortized, random.

Universal Hashing:
- Even better \(\rightarrow\) randomize!
- Let \(H\) be a family of hash fns
  \Select h \in H randomly
- If \(x \neq y\) then \(\Pr(h(x) = h(y)) = \frac{1}{m}\)
  \(h(x)\) \(\text{random} \pmod{p}\)

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:
  \[ \lambda = \frac{n}{m} \]
- Running times increase as \(\lambda \to 1\)
- Hash function:
  \[ h: \text{Keys} \to [0..m-1] \]
  \(\rightarrow\) Should scatter keys random.
  \(\rightarrow\) Need to handle collisions
  \(h(x) = h(y)\)

Hashing I

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

Eg. Java variable names:
- temp1, temp2, temp3

Why \(\text{\text{mod p \mod m}}\)?
- Modding by a large prime scatters keys
- \(m\) may not be prime
  (e.g., power of 2)

Common Examples:
- Division hash:
  \(h(x) = x \mod m\)
- Multiplicative hash:
  \(h(x) = (ax \mod p) \mod m\)
- Linear hash:
  \(h(x) = ((ax + b) \mod p) \mod m\)

Assume keys can be interpreted as ints
Overview:

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

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

If \(\lambda < \lambda_{\text{min}} \text{ or } \lambda > \lambda_{\text{max}}\) ? Rehash!

- Alloc. new table size = \(\nicefrac{n}{\lambda_o}\)
- Compute new hash fn \(h\)
- Copy each \(x,v\) from old to new using \(h\)
- Delete old table

Separate Chaining:

- table[i] is head of linked
  - list of keys that hash to \(i\).

Example:

<table>
<thead>
<tr>
<th>Keys ((x))</th>
<th>(h(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>t f u p z d</td>
<td>5 4 1 2 3</td>
</tr>
</tbody>
</table>

Analysis: Recall

- Load factor \(\lambda = \frac{n}{m}\)
- \(n\) = # of keys
- \(m\) = table size

Thm: Amortized time for rehashing

\[1 + \left(2\lambda_{\text{max}} / (\lambda_{\text{max}} - \lambda_{\text{min}})\right)\]

Thm: Expected search time

\[S_{\text{sc}} = 1 + \frac{\lambda}{2}\]

Thm: Expected search time

\[U_{\text{sc}} = 1 + \lambda\]

How to control \(\lambda\)?

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

Designer: \(\lambda_{\text{min}}, \lambda_{\text{max}}\) - allowed

\(\lambda_o = \frac{\lambda_{\text{min}} + \lambda_{\text{max}}}{2}\) - ideal

Proof: On avg. each list has \(\frac{n}{m} = \lambda\)

- Success: 1 for head + half the list
- Unsuccess: 1 " " + all the list

If \(\lambda < \lambda_{\text{min}} \text{ or } \lambda > \lambda_{\text{max}}\) ...
Open Addressing:
- Special entry ("empty") means this slot is unoccupied
- Assume $\lambda \leq 1$
- To insert key:
  - check: $h(x)$ if not empty try $h(x)+i_1$
  - $h(x)+i_2$
  - $h(x)+i_3$
  - 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?

Hashing III

Linear Probing:
- $h(x), h(x)+1, h(x)+2, ...$
- until finding first available

Simple, but is it good?
- $x: d, z, p, w, t$
- $h(x): 0, 2, 2, 0, 1$
- $t$ did not collide directly but had to probe 3 times!

Open Addressing

Analysis:
- Let $S_{LP}$ = expected time for successful search
  - $U_{LP} = $ "unsuccessful"

Let $S_{LP} = \frac{1}{2} (1 + \frac{1}{1-\lambda})$
- $U_{LP} = \frac{1}{2} (1 + \frac{1}{1-\lambda})^2$
- Obs: As $\lambda \to 1$ times increase rapidly

Analysis: Improves secondary clustering
- May fail to find empty entry
  - (Try $m=4$, $j^2 \mod 4 = 0 \iff 1 \text{ but not } 2 \equiv 3$)
- How bad is it? It will succeed if $\lambda < \frac{1}{2}$.

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

Pf: See latex notes.

Clustering:
- Clusters form when keys are hashed to nearby locations
- Spread them out?

Quadratic Probing:
- $h(x), h(x)+1, h(x)+4, h(x)+9, ...$
- $h(x)+t_1 + t_2 + t_3$
- wrap around if $t > m$
Double Hashing:
(Best of the open-addressing methods)
- Probe sequence det’d by second hash fn. - g(x)
  \( h(x) + \{0, g(x), 2g(x), 3g(x)\ldots\} \mod m \)

Recall:
- Separate Chaining:
  Fastest but uses extra space (linked list)
- Open Addressing:
  Linear probing: clustering
  Quadratic probing:
  \( h(x) \)
  \( g(x) \)
  \( g(x) \)
  \( g(x) \)
  (until finding an empty slot)

Why does bust up clusters?
Even if \( h(x) = h(y) \) [collision]
it is very unlikely that \( g(x) = g(y) \)
\( \Rightarrow \) Probe sequences are entirely different!

Analysis: Defs:
\( E_{DH} \) = Expected search time of double hash. if successful
\( U_{DH} \) = Exp. if unsuccessful
Recall: Load factor \( \lambda = n/m \)

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

\( \rightarrow \) Proof is nontrivial (skip)

Delete \( x \): Apply find(\( x \))
- Not found \( \Rightarrow \) error
- Found \( \Rightarrow \) set to "empty"

Problem:
insert \( x \): \( h(x) \)
delete \( x \): \( h(x) \)
find \( x \): \( h(x) \)

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

\( \text{find}(x), h(x) \rightarrow \text{Not found!} \)

Dictionary Operations:

Insert \( (x,v) \): Apply probe sequence until finding first empty slot.
- Insert \( (x,v) \) here.
  (If \( x \) found along the way \( \Rightarrow \) duplicate key error!)
Multiway Search Trees:

- 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 \( \lceil \frac{m}{2} \rceil \) 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

Node Structure: constant int \( M \):
```java
class BTreeNode {
    int nChild; // no. of children
    BTreeNode child[M]; // children
    Key key[M-1]; // keys
    Value value[M-1]; // values
}
```

Example: \( m = 5 \)
[Each node has:
3-5 children
2-4 keys]

Theorem: A B-tree of order \( m \) with \( n \) keys has height at most \( (\log n)/\gamma \), where \( \gamma = \log(\lfloor m/2 \rfloor) \)
(See full notes for proof)
Key Rotation (Adoption)
- A node has too few children  \([m/2] - 1\)
- Does either immediate sibling have extra? \(\geq [m/2]+1\)
- 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 \(m' = [m/2] \) and \(m'' = m+1-[m/2]\)

**Lemma:** For all \(m \geq 2\),
\([m/2] \leq m+1-[m/2] \leq m\)
\(\Rightarrow m' + m'' \) are valid node sizes

B-Tree restructuring:
- Generalizes 2-3 restructuring
- Key rotation (Adoption)
- Splitting (insertion)
- Merging (deletion)

B-Tree restructuring:
- Generalizes 2-3 restructuring
- Key rotation (Adoption)
- Splitting (insertion)
- Merging (deletion)

Node Merging:
- A node has too few children \([m/2]-1\)
- Neither sibling has extra (both \([m/2]\)"
- Merge with either sibling to produce node with \(([m/2]-1)+[m/2]\) children
**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

**Deletion:**
- Find key to delete
- Find replacement/copy
- If underfull (\[\frac{m}{2}\]-1) child
  → If sibling can give child
    → Key rotation
  → Else (sibling has \[\frac{m}{2}\])
    → Merge with sibling
  → Propagates: If root has 1 child, collapse root

**Example:** \( m = 5 \)

**Example:**
- Insert(29)
- Split
- Promotes key
- Key rotation

**Example:** \( m = 5 \)
- Delete(30)
- Merge
- Key Rotation

**B-Trees III**
History:
1989: Seidel & Aragon
[Explosion of randomized algorithms]
Later discovered this was already known: Priority Search Trees from different context (geometry) McCreight 1980

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

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
(Std. BST) \( 1 \) 2 3 4 5 6 7
(Treap) \( 2 \) e \( 5 \) \( 4 \) 6 4

Along any path - Insertion times increase

Geometric Interpretation:

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

Trea: 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.

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

**Proof:** Follows directly from BST analysis

**Implementation:** (See pdf notes)
- Node: Stores priority + usual...
- Helpers:
  - lowest priority ($p$) returns node of lowest priority among:
  - restructure: performs rotation $p.left$ (if needed) to put lowest priority node at $p$.

**Example:**

[Diagram of treap operations and analysis]
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”)

Tries and Digital Search Trees I

Digital Search:
- Keys are strings over some alphabet \( \Sigma \)
- Eq. \( \Sigma' = \{ a, b, c, \ldots \} \)
- Assume chars coded as ints: \( a = 0, b = 1, \ldots z = k - 1 \)

Example:

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
  - \( x \) \( \neq x \) \( \Rightarrow \) try next char in \( \Sigma' \)
    \( = x \) \( \Rightarrow \) advance to next character of search string
  - First child/next-sibling

Same structure/Alt. Drawing

Search: \( \sim \) length of query string \( [O(1) \text{ time per node}] \)

Space:
- No. of nodes \( \sim \) total no. of chars in all strings
- Space \( \sim k \cdot (\text{no. of nodes}) \)
Dealing with long Paths:
- To get both good space and query time efficiency, need to avoid long, degenerate paths.

Path compression!

Tries and Digital Search Trees II

Example:
- Def: Substring identifier for $S_i$: is shortest prefix of $S_i$ unique to this string
- $S_{i-1}$ - a.m.
- $S_i$ - a.m.a.

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

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

Q: What is minimum substring needed to identify suffix $S_i$?
Example: \( S = \text{pamapajama} \)

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

Try and Digital Search Trees III

Analysis:
- Space: \( O(n) \) nodes
- \( O(n \cdot k) \) total space
  \( (k = |\Sigma| = o(1)) \)
- Search time: \( n \) total length of target string
- Construction time:
  - \( O(n \cdot k) \) [nontrivial]

Example:
- Search("ama") \( \rightarrow \) End at intern node \( \rightarrow \) amap
- Report: 2 occ. \( \leftarrow \) amap
- Search("amapaj") \( \rightarrow \) End at extern node
- Go to \( S_i \) + verify

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

\( \left\lfloor \frac{x}{2} \right\rfloor \) stop when each cell has < 2 points

Claim: This is a trie!
Binary Encoding:
- Assume our points are scaled to lie in unit square 0 ≤ x, y < 1 (can always be done)
- Represent each coordinate as binary fraction:
  \[ x = 0.a_1a_2a_3,... \quad a_i \in \{0,1\} \]
  \[ x = \mathcal{T}_i:a_i \cdot \frac{1}{2^i} \]

Example:

<table>
<thead>
<tr>
<th>0.0</th>
<th>0.01</th>
<th>0.1</th>
<th>1 base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.01</td>
<td>0.1</td>
<td>1 base 2</td>
</tr>
</tbody>
</table>

How do we extend to 2-D?

PR kd-Tree ≗ Trie ??
- Approach: Show how to map any point in \( \mathbb{R}^n \) to bit string
  - Store bit strings in a trie (alphabet \( \mathcal{E}_r = \{0,1\} \))
  - Prove that this trie has same structure as kd-tree

Tries and Digital Search Trees IV

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_2a_3...
Y = 0.b_1b_2b_3...
Z = 0.c_1c_2c_3...

Lemma: Given a pt set \( P \subseteq \mathbb{R}^2 \) (in unit square \( [0,1]^2 \)) let
\( P = \{p_1, ..., p_n\} \) where \( p_i = (x_i, y_i) \)
Let \( \Phi(P) = \{ \Phi(p_1), \Phi(p_2), ..., \Phi(p_n) \} \)
(n binary strings)
Then the PR kd-tree for \( P \) is equivalent to binary trie for \( \Phi(P) \).

Bit Interleaving:
Given a point \( p = (x, y) \)
  0 ≤ x, y < 1
let: \( x = 0.a_1a_2a_3... \) in binary
  \( y = 0.b_1b_2b_3... \)
Define:
\( \Phi(x, y) = a_1b_1a_2b_2a_3b_3... \)
Called Morton Code of \( p \)

Proof: By induction on no. of bits
Let \( x = 0.a_1a_2... \quad y = 0.b_1b_2... \)
and consider just \( \Phi(x, y) = a_1b_1a_2b_2a_3b_3... \)

\( a_1 = \)
\( b_1 = \)
\( a_2, b_2 = \)
\( a_3, b_3 = \)

The PR kd-tree + binary trie assign pts to same subtrees
(...) induction
Deallocation Models:
- **Explicit**: (C++)
  - programmer deletes
  - may result in leaks
- **Implicit**: (Java, Python)
  - runtime deletes
  - Garbage collection
  - Slower runtime
  - Better memory compaction

Explicit Allocation/Deallocation
- Heap memory is split into blocks whenever requests made
- Available blocks:
  - merged when contiguous
  - stored in available block list

What happens when you do
- `new` (Java)
- `malloc/free` (C)
- `new/delete` (C++)?

Runtime System Mem. Mgr.
- Stack - local vars, recursion
- Heap - for "new" objects
  - Don't confuse with heap data structure/heapsort

Block Structure:
- **Allocated**
  - `inUse`, `prev`, `inUse`
  - links in avail. list
  - total block size (includes headers)

- **Available**
  - `prev`, `next`
  - stored in available block list

Memory Management

Guide:
- `prevInUse`: 1 if prev. contiguous block is allocated
- `prev/next`: links in avail. list

How to Select from available blocks?
- **First-fit**: Take first block from avail. list that is large enough
- **Best-fit**: Find closest fit from avail. list

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

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

External: Caused by pattern of alloc/dealloc
Internal: Induced by mem. manage. policies (not user)
Example: Alloc \( b = 59 \)

Allocation: \( \text{malloc}(b) \)
- Search avail. list for block of size \( b' \geq b+1 \)
- If \( b' \) close to \( b \): alloc entire block (unlink from avail list)
- Else: split block

Deallocation:
- If prev\(+next\) contiguous blocks are allocated \( \rightarrow \) add this to avail
- Else \( \rightarrow \) merge with either/both to make max. avail block

Example:

```c
(int) realloc((void*) ptr, size); // new block
```

Memory Management

Some C-style pointer notation
- void* - pointer to generic word of memory

Let \( p \) be of type void*:
  \( p+10 \) - 10 words beyond \( p \)
  \( *(p+10) \) - contents of this

Let \( p \) point to head of block:
  \( p->\text{inUse}, p->\text{prevInUse}, p->\text{size} \)
  - We omit bit manipulation
  \( *(p+p->\text{size}-1) \) - references last word in this block

```c
(void**) alloc (int b) {
    b += 1 // add 1 for header
    p = search avail list for block of size \( b \)
    if (p == null) Error: Out of memo!
    if (p->size - b < TOO_SMALL)
       unlink p from avail. list
       \( q = p \)
    else ... (continued)
}
```

```c
p->size -= b // remove allocation
*(p+p->size-1) = b // size 2
q = p + p->size // start of new block
q->size = b
q->prevInUse = 0 // new block header
q->inUse = 1
(q + q->size).prevInUse = 1 // update prevInUse for next contig. block
return q+1 // skip over header
```
**Buddy System:**
- Block sizes (including headers) are powers of 2.
- Requests are rounded up (internal fragmentation).
- Block size $2^k$ starts at address that is multiple of $2^k$.
- $k =$ level of a block.

**Structure:**

<table>
<thead>
<tr>
<th>Level</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>15</td>
</tr>
</tbody>
</table>

**In practice:** There is a minimum allowed block size. Buddy system only allows allocations aligning with these blocks.

**Coping With External Fragmentation**
- Unstructured allocation can result in severe external fragmentation.
- Can we compress? Problem of pointers.
- By adding more structure, we can reduce external fragmentation at cost of internal fragmentation.

**Example:**
- `alloc(2)` rounds up → `alloc(4)`
- Split into smaller blocks.
- `return`.

**Allocation:** `alloc(6)`
- $k = \lceil \lg (b+1) \rceil$
- If `avail[k]` non-empty - return entry + delete.
- Else: find `avail[j] ≠ ∅` for $j > k$.
- Split this block.

**Merging:**
- When two adjacent blocks are available, we don't always merge them.
- Must have same size: $2^k$.
- Must be buddies - siblings in tree structure.

**Def.:**
- `buddy_k(x) = \{ x + 2^k \text{ if } 2^k \text{ divides } x \}
- \{ x - 2^k \text{ otherwise} \} + x\text{ [Bit manipulation]}

**Big Picture:**
- `avail` list is organized by level: `avail[k]`.
- Block header structure same as before except:
  - `prevInUse` if not needed size 2.
**False Positive:** We report \( x \in X \), but it is not.

**False Negative:** We report \( x \notin X \), but it is.

We will tolerate false positives (very few) but no false negatives.

**Examples:**

\[ X = \{ \text{weak passwords} \} \]

- \( \rightarrow \) Screen passwords for safety
- \( \rightarrow \) Allow no weak passwords
- \( \rightarrow \) May flag a few good passwords as being weak.

\[ X = \{ \text{URL's of malicious web sites} \} \]

- \( \rightarrow \) Allow no malicious sites
- \( \rightarrow \) May flag a few safe ones

**Bloom Filter:**

- 1970 by Burton Howard Bloom
- Can store very large sets \( X \)
- Answers queries in \( O(1) \) time
- Uses \( O(n) \) bits - \( n = |X| \)

\( (\text{may be smaller than space needed to store all keys!} \)

**Filtering:**

- Given a large set \( X \) of keys answer membership queries is \( x \in X \)?

**Objectives:**

- Fast! \( O(1) \) time
- Yes/No: No values/Just keys
- Errors allowed

**Bloom Filter:**

Let \( X \subseteq U \) (universe )

\[ |X| = n \text{ large!} \]

**Parameters:** \( k,m \) - TBD

**Bit Vector:** \( B[0..m-1] \)

**K Hash Functions:** \( h_1, \ldots, h_k \)

\[ h_i : U \rightarrow \{ 0, \ldots, m-1 \} \]

[Think: \( h_i \) maps keys to random locations in bit vector]

**Find (Key \( x \))**

\[
\text{for } (i = 1, 2, \ldots, k) \}
\[
\quad \text{if } (B[h_i(x)] == 0) \text{ return false}
\]
\[
\quad \text{return true}
\]

**Find ("b") \( h_i(b) = [2,3,7] \rightarrow \text{true} \)

**Find ("c") \( h_i(c) = [2,5,9] \rightarrow \text{false} \)

**Find ("d") \( h_i(d) = [0,4,7] \rightarrow \text{true?} \)

**Example:**

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 2 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

**Insert ("a") \( h_i(a) = [0,7,4] \)

**Insert ("b") \( h_i(b) = [2,3,7] \)

(all other entries 0)

**Initially:** \( B[i] = 0, i = 0, \ldots, m-1 \)

**Insert (Key \( x \))**

\[
\text{for } (i = 1, 2, \ldots, k) \}
\[
\quad B[h_i(x)] \leftarrow 1 
\]

\[
\]
Controlling False Positives:

**Obs:**
- m - Bigger is better (but more storage)
- k - Trickier to balance

**Math Facts:**
- If \(|x| \leq 1\), then \(1 - e^{-x} \approx e^x\)
- If an event occurs with probability \(p\), the probability of \(k\) independent occurrences is \(p^k\)

**Analysis:**
- Assume \(n = |X|, m = |B|, k = \text{no. hashes}\)
- Prob of hitting an arbitrary loc. of \(B\), \(B[j]\), with any hash is \(1/m\).
- Prob of missing is \(1 - 1/m\) so:
  \[
  \Pr(B_{i}(x) \neq j) = 1 - 1/m
  \]
- After inserting all \(n\) keys, each with \(k\) hashes, prob of missing \(B[j]\) is:
  \[
  \Pr(B[j] = 0) = (1 - 1/m)^{kn}
  \]
- Assuming \(m\) large \(\Rightarrow |x| \leq 1\) small \(\Rightarrow\)
  \[
  \Pr(B[j] = 0) \approx (e^{-1/m})^{kn} = e^{-kn/m}
  \]

**Partial Correctness:**
- If \(x \in X\), all hash locations \(B[h_i(x)]\) set to 1 \(\Rightarrow\) true
- If \(x \notin X\), any hash loc. \(B[h_i(x)]\) is 0 \(\Rightarrow\) false
  \[
  \rightarrow\text{But by coincidence, all may be set to 1 by other keys} \Rightarrow \text{true (false positive)}
  \]

**Bloom Filters II**

**In summary:**
\[
\Pr[FP] = \left(\frac{1}{2}\right)^{\ln \frac{m}{n}}
\]

To achieve a false pos. rate of \(d > 0\) set
\[
\frac{m}{n} \approx \frac{\log \frac{1}{d}}{\ln 2} = O(\log \frac{1}{d})
\]

**False Positive Probability:**
- False positive (FP) occurs if all \(h_i(x)\) set to 1 by other keys
- Since prob \(B[j] = 0\) is \(p\), we have
  \[
  \Pr[FP] = \Pr[B[h_i(x)] = 1]^k = (1 - p)^k
  \]
- To simplify: Take \(ln\)
  \[
  \ln(\Pr[FP]) = k \cdot \ln(1-p)
  \]
- By def. of \(p\), we have:
  \[
  \ln p = -kn/m \Rightarrow k = -\frac{m}{n} \ln p
  \]
  \[
  \ln(\Pr[FP]) = -\frac{m}{n} \ln p \cdot \ln(1-p)
  \]
- How to set \(k\) to minimize this?
  \[
  \text{Assume } m, n \text{ fixed, but } p \text{ varies}
  \]
  \[
  \ln p \cdot \ln(1-p) \rightarrow \frac{m}{n} \ln p \cdot \ln(1-p)
  \]
- Set \(p = \frac{1}{2}\)
  \[
  \Pr[FP] = (1 - p)^{k} = \left(\frac{1}{2}\right)^{\frac{m}{n} \ln 2} \approx 0.608
  \]
Decrease-key:
- Given an entry \((x, v)\), decrease the key value from \(x\) to \(y\).
- How to identify the entry?
  - Heaps do not support an efficient way to find keys.

Locator: A special (abstract) object that identifies an entry of the heap.

Locator \(r = \text{insert}(x, v)\):
- \(\text{decrease-key}(r, y)\)

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

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

Why decrease-key?
- Dijkstra's algorithm
  - Heap tracks distances to vertices from source
  - \(n\) extract-mins
  - \(upto n^2\) decrease-keys
  - Want decrease key fast!

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

1984: Fibonacci Heaps (Fredman + Tarjan)
- Many variants
- Complex to analyze

2013: Quake Heap (Timothy Chan)
- Much simpler

Locator is a public object.
- Dijkstra's algorithm
  - Heap tracks distances to vertices from source
  - \(n\) extract-mins
  - \(upto n^2\) decrease-keys
  - Want decrease key fast!

Quake Heaps I
- Basic definitions
- Operations

History:
- 1984: Fibonacci Heaps (Fredman + Tarjan)
- Many variants
- Complex to analyze

2013: Quake Heap (Timothy Chan)
- Much simpler
Assuming we have Basic utilities:

`void decrease-key(locator r, Key y)`

Node $u = r$.get Node() // get leaf node
Node $u$.child = null

```java
    do {
        u.key = y // update key value
        uChild = u; u = u.parent // go up
    } while (u != null & u.child == u.left)
    if (u != null) cut(u) // cut subtree
```

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

### Insert: Super lazy! Just make a single node tree

### Locator insert($key$ $x$)

Node $u$ = new trivial-tree($x$) return new Locator($x$)

---

**Quake Heaps II**
- Utility ops
- Insert
- Decrease-key

We'll apply these utilities to implement operations

**Basic utilities:**

- `void make-root(Node u)`
  - $u$.parent = null
  - add $u$ to roots[$u$.level]

- `cut(Node w)`
  - Assuming $w$ has right child - cuts it off as new root

- `Node trivial-tree(Key $x$)`
  - Node $u$ = new Node(key $x$, level 0)
  - nodeC[t][0] += 1
  - make-root($u$)
  - return $u$

- `Node link(Node u, Node v)`
  - int lev = $u$.level + 1 (== $v$.level + 1)
  - if ($u$.key = $v$.key)
    - $w$ = new Node($u$.key, lev, $u$, $v$)
  - else $w$ = new Node($v$.key, lev, $v$, $u$)
  - nodeC[t][lev] += 1
  - $u$.parent = $v$.parent = $w$
  - return $w$

- `void cut(Node w)`
  - Node $v$ = $w$.right
  - if ($v$ != null)
    - $w$.right = null
    - make-root($v$)

- `make-root(Node u)`
  - Make $u$ a root

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

- `link(Node u, Node v)`
  - Link $u$ + $v$
  - $u$ + $v$ roots on same level

- `smallest-key-left-child`

---

**Node link(Node u, Node v)**

- `int lev = u.level + 1 (== v.level + 1)`
- `if (u.key = v.key)`
  - `w = new Node(u.key, lev, u, v)`
- `else w = new Node(v.key, lev, v, u)`
  - `nodeC[t][lev] += 1`
  - `w.parent = v.parent = w`
  - `return w`
Extract Min:
- Find the root with smallest key (brute force)
- Delete all nodes down to leaf - many trees
- Merge trees together
  - Work bottom-up
  - Merge 2 trees at level k to form tree at level k+1
- Too 'stringy'? Flatten QUIKE!

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

Intuition: Tree becomes "stringy" after many extractions.
- This is evidenced by the fact that node counts do not decrease
- When this happens - we flatten so we can build up later

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

Finally, return 4
**Key: extract-min()**

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

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

**Void delete-left-path(u)**

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

**Void merge-trees()**

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

**Void quake()**

- **for** (lev = 0..nLevels - 2)
  - **if** (nodeCt[lev+1] > \(\frac{3}{4}\) nodeCt[lev])
  - **clear-all-above(lev)**

**Decrease-key:**
- **Locate leaf node** - \(O(1)\)
- **Trace path up left-child links**
- **Cut** \(O(1)\)
- **Change key** \(O(height) = O(log n)\)

**Quake Heaps IV**
- **Extract min (cont)**
- **Faster decrease key**

**Decrease-key:**
- Locate leaf node - \(O(1)\)
- Trace path up left-child links
- Cut \(O(1)\)
- Change key \(O(height) = O(log n)\)

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

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

**Insert:** \(O(1)\)
**Decrease-key:** \(O(log n)\)
**Extract-min:** ??
Will show \(O(log n)\) amortized
**Amortized Analysis**

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

**Potential-Based Analysis**

- Each instance of the data structure assigned a potential $\Psi$.
- Low potential $\Rightarrow$ good structure.
- High potential $\Rightarrow$ bad structure.

**Why is Quake Heap efficient?**

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

**Intuition:**

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

**Potential decrease compensates for high actual cost.**

**Quake Heaps V**

- Analysis (Quick + Dirty)

**Lemma:** Amortized cost of `insert`, `dec-key` $= O(1)$

```
extract-min = O(\log n)
```

**Quake Heap Potential:**

Let $N = \text{no. of nodes}$

$$\Psi = N + 2R + 4B$$
Minimum Spanning Trees:
- Given a connected, weighted graph $G=(V,E)$
  $(u,v)\in E \rightarrow w(u,v) =$ weight
- Spanning Tree:
  - A subset $T \subseteq E$ of edges that connect all the vertices and is acyclic
  - Total weight: $w(T) = \sum_{(u,v) \in T} w(u,v)$

Minimum Spanning Tree (MST):
- A spanning tree of min. weight

Data Structures & Algorithm Design:

- Euclidean Min. Spanning Tree (II)

Euclidean Graph:
Given a set $P=\{p_1, \ldots, p_n\}$ of pts in $\mathbb{R}^d$, this is a complete graph (all $\binom{n}{2}$ edges)
where:
$$w(p_i,p_j) = \text{dist}(p_i,p_j) = \sqrt{(x_i-x_j)^2 + (y_i-y_j)^2}$$

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

How are data structures used?
- Transaction/Query:
  - Insert new student
    - name = "Mary" ID = 1234...

  - Closest coffee to my location

  - Algorithms:
    - Dijkstra - Fibonacci Heap
    - Kruskal - Union/Find

Algorithms for MSTs:
- Based on greedy construction
- Add the lightest edge that causes no cycle

Kruskal's, Prim's, Borůvka's

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

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

Euclidean MST (EMST):
- The MST of $P$'s Euclidean graph
Finding next edge?
- Brute force: $O(n^2) \Rightarrow O(n^3) \bigcirc$
- kd-tree: To compute near neighbor
- Priority queue: To find best pair

Prim's Algorithm:
- Given point set $P$, start pt $s$
- $S \subseteq P$: Pts in spanning tree
  - Init: $S = \{s\}$ End: $S = P$
  - $P \setminus S$: Pts not yet in tree

  while ($S \neq P$)
  - find closest $(p_i, p_j)$ s.t. $p_i \in S$ and $p_j \in P \setminus S$
  - add $p_i$ to $S$
  - add $(p_i, p_j)$ to tree

Prim (Points $P$, Point start)
- Initialize (later)
  - add start to in EMST
  - $nn \leftarrow$ kdTree. nearNeigh (start)
- Add start to dep [nn]
  - Add new NN pair (start, nn)
while (kdTree $\neq \emptyset$)
  - edge $\leftarrow$ heap. extractMin()
  - if (edge.getSecond() $\in$ inEMST)
    - Add Edge (edge) (later)

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

Priority Queue: Stores the NN pairs ordered by squared dist.

- List: Store edges of tree
  - (e.g. [{SFO, DFW}, {DFW, ORD}, ..])
- Set: Store points of $S$
  - (e.g. [{SFO, DFW, ORD, ATL}])
- Spatial Index: Stores pts of $P \setminus S$. Answers NN queries

Hash map of lists: Stores dependency lists, indexed by point

Nearest-Neighbor Pairs:
- Given $p_i \in S$, let $p_j$ be the closest point in $P \setminus S$
  - $(p_i, p_j)$ is nearest-neighbor pair
- Given NN pair $(p_i, p_j)$
  - We say $p_i$ depends on $p_j$

Dependents list $\text{dep}(p_j)$ is list of all pts $p_i$ that depend on $p_j$

<table>
<thead>
<tr>
<th>Point $p$</th>
<th>dep($p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWI</td>
<td>{SFO}</td>
</tr>
<tr>
<td>DCA</td>
<td>{DFW}</td>
</tr>
<tr>
<td>SEA</td>
<td>{}</td>
</tr>
<tr>
<td>IAD</td>
<td>{ORD, ATL}</td>
</tr>
<tr>
<td>LAX</td>
<td>{}</td>
</tr>
</tbody>
</table>
addEdge(Pair<Point> edge) △
add edge to edgeList (first, second)
pt2 ← edge.getActiveSecond()
add pt2 to inEMST
delete pt2 from kdTree
dep2 ← get pt2 dep list from dependents
for each (pt3 in dep2)
  mn3 ← kdTree. nearNeighbor(pt3)
  if (mn3 == null) break
  add NN(pt3, mn3)

Helpers:
- Initialize(Point start)
  - initialize all structures
  - addEdge(Pair<Point> edge)
    - add new edge to EMST
  - add NN(Point pt, Point nn)
    - add new NN pair (pt, nn)

Euclidean MSTs (III)

Q: Why check mn3 == null?
- On adding last pt to EMST the kd-tree is empty.

add NN(Point pt, Point nn)
  dist ← distanceSq(pt, nn)
  pair ← new Pair(pt, nn)
  insert pair in heap w. priority dist
  add pt to dep[nn]

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

Thats it!

Is this efficient?
- Assuming NN queries in O(log n) time

Total time = O(Cn·log n + m·log n)

m = # of NN updates
Much depends on m. m depends on pt. distrib.
**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

**Example:**
- Insert:
  - same as standard BST
  - if depth too high
    - trace search path
    back
  - find unbalanced
    node — scapegoat
  - rebuild this subtree
- Delete:
  - Same as std. BST
  - If num. of deletes is large rel. to $n$ —
    rebuild entire tree!
- Scapegoat Trees

**Overview:**
- Insert:
  - same as standard BST
  - if depth too high
    - trace search path
    back
  - find unbalanced
    node — scapegoat
  - rebuild this subtree
- Delete:
  - Same as std. BST
  - If num. of deletes is large rel. to $n$ —
    rebuild entire tree!

**Find:** Same as std. BST
- Tree height $\leq \log_{3/4} n \approx 1.71 \log n$

**How to rebuild?**
- Inorder traverse $p$’s subtree $ightarrow$ array $A[]$
- buildSubtree($A$)

$buildSubtree(A[0..k-1])$:
- if $k = 0$ return null
- $j = \lceil k/2 \rceil$ ; $x = A[j]$ median
- $L \leftarrow buildSubtree(A[0..j-1])$
- $R \leftarrow buildSubtree(A[j+1..k-1])$
- return Node($x$, $L$, $R$)
Details of Operations:

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, \( \text{size}(p) = \text{no. of nodes in } p \)'s subtree \}
  - if \( \frac{\text{size}(p.\text{child})}{\text{size}(p)} > \frac{2}{3} \)

\( p \); rebuild (p)
  - break

Scapegoat Trees II

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

Example:

Insert (s)

Time:
\( O(n) \)

How to compute \( \text{size}(p) \)?
- Can compute it on the fly
- While backing out, traverse "other sibling"
- Too slow? No! \( \Rightarrow \) Charge to rebuild

Proof: By contradiction

Suppose \( p \)'s depth \( > \log_{3/2} n \) but \( \forall \) ancestors

Lemma: Given a binary tree with \( n \) nodes, if
\( \exists \) node \( p \) of depth \( > \log_{3/2} n \), then \( \exists \) ancestor of \( p \) that satisfies scapegoat condition
**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
  \[ \rightarrow \text{Amortize rebuild time against delete ops} \]
- **Insert:** Based on potential argument
  \[ \rightarrow \text{It takes } n \sim k \text{ ops to cause a subtree to size } k \text{ to be unbalanced.} \]
  \[ \rightarrow \text{Charge rebuild time to these operations} \]
**Range Trees:**
- Space is $O(n \log^d n)$
- Query time:
  - Counting: $O(\log^d n)$
  - Reporting: $O(k + \log^d n)$
  - In $\mathbb{R}^2$: $\log^2 n$ much better than $\log n$ for large $n$

**kd-Tree:**
- General-purpose data structure for pts in $\mathbb{R}^d$
- Orthogonal range query:
  - Count/report pts in axis-aligned rect.
  - Answer: $O(\log^d n)$
  - Count/report pts in axis-aligned rect.
    - No. of pts reported

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

**Layering:** Combining search structures
- Suppose you want to answer a composite query w. multiple criteria:
  - Medical data: Count subjects
    - Age range: $a_o \leq \text{age} \leq a_i$
    - Weight range: $w_o \leq \text{weight} \leq w_i$
  - Design a data structure for each criterion individually
  - Layer these structures together to answer full query

**1-Dim Range Tree:**
- $d = 1$
- $Q_{i_0}$ to $Q_{i_n}$
- $\text{Count} = 1 + 2 + 2 + 4 + 1 = 10$
- $Q_{i_0}$ to $Q_{i_n}$

**Canonical Subsets:**
- Goal: Express answer as disjoint union of subsets
- Method: Search for $Q_{i_0} + Q_{i_1}$ and take maxima/subtrees

**Range Trees I**
- Multi-Layer Data Structures
Recursive helper:
\[
\text{int } \text{range}1Dx(\text{Node } p, \text{Intv } Q = [Q_o, Q_u], \text{Intv } C = [x_o, x_i])
\]

Initial call: range1Dx(root, Q, C)

Cases:
1. **p is external:**
   - If \( p pt. x \in Q \) → 1 else → 0
2. **p is internal:**
   - \( C \subseteq Q \) → all of p's pts lie within query
     → return p.size
   - \( C \text{ is disjoint from } Q \) → 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 \( Q = [Q_o, Q_u] \) be query interval
- For each node \( p \), define interval cell \( C = [x_o, x_i] \)
  s.t. all pts of p's subtree lie in C
- Root cell: \( C_o = [-\infty, +\infty] \)

Range Trees II

2-D Range Searching:
- **Layer** a range tree for x with range tree for y
- For each node \( p \in 1D \cdot x \) tree, let \( S(p) = \text{set of pts in p's subtree} \)
- 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
\[ Q = [Q_{lo}, Q_{hi}] \times [Q_{lo}, Q_{hi}] \]
- Run range 1Dx to find all subtrees that contribute
  - For each such node p,
    - Run range 1Dy on p.aux
  - Return sum of all result

2D Range Tree:
- Construct 1D range tree based on x coord for all pts
- For each node p:
  - Let \( S(p) \) be pts of pi tree
  - Build 1D range tree for \( S(p) \) based on y \( \rightarrow \) p.aux
- Final structure is union of \( x \)-tree + \((n-1)\) y-trees

Higher Dimensions?
- In d-dim space, we create d-layers
- Each recurses one dim lower until we reach 1-d search
- Time is the product:
  \[ \log n \cdot \log n \cdot \ldots \cdot \log n = O(\log^d n) \]

Analysis: The 1D x search takes of \( O(\log n) \) time and generates \( O(\log n) \) calls to 1Dy search
\[ \Rightarrow \text{Total: } O(\log n \cdot \log n) = O(\log^2 n) \]

Analysis:
- Invoked \( O(\log n) \) times once per maximal subtree
- Invoked \( O(\log n) \) times once for each ancestor of max subtree

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

```c
int range2D(Node p, Rect Q, InvPt C = [x0, x1]) {
  if (p is external) return p.p.t \( \notin \) Q
  else if (Q \( \times \) contains C) // C \( \subseteq \) Q, x-projection
    \[ [y_0, y_1] = [\infty, \infty] \] // init y-cell
    return range1Dy(p.aux, Q, [y_0, y_1])
  else if (Q.x is disjoint of C) return 0
  else // partial x-overlap
    return range2D(p.left, Q, [x0, p.x]) + range2D(p.right, Q, [p.x, x1])
}
```