Data structures are **FUNDAMENTAL!**

- All fields of CS involve storing, retrieving and processing data
- Information retrieval
- Geographic Info. 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

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

**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?

**Common:**
- \( O(1) \): constant time 😊
  - [Hash map]
- \( O(\log n) \): log-time (good)
  - [Binary search]
- \( O(n^p) \): \( p \) = constant; poly time
  - \( O(n\log n) \)

**Asymptotic: "Big-o"**
- Ignore constants
- Focus on large \( n \)
- \( T(n) = 34n^2 + 15n\log n + 143 \)
- \( T(n) = O(n^2) \)

**Asymptotic Analysis:**
- Run time as function of \( n \): no. of items
- Worst-case, average case, randomized...
- Amortized - average over series of ops.
Linear List ADT:
Stores a sequence of elements \( a_1, a_2, \ldots, a_n \). Operations:
- \text{init()} - create an empty list
- \text{get(i)} - returns \( a_i \)
- \text{set(i, x)} - sets \( i \)th element to \( x \)
- \text{insert(i, x)} - inserts \( x \) prior to \( i \)th
   (moving others back)
- \text{delete(i)} - deletes \( i \)th item
   (moving others up)
- \text{length()} - returns num. of items

Implementations:
- Sequential: Store items in an array
  \[
  a_1, a_2, \ldots, a_n
  \]
- Linked allocation: linked list
  - Singly: \( a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{null} \)
  - Doubly: \( a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{null} \)

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

Doubling Reallocation:
- When array of size \( n \) overflows
  - allocate new array size \( 2n \)
  - copy old to new
  - remove old array

Dynamic Lists + Sequential Allocation: What to do when your array runs out of space?
- Deque ("deck"): Can insert or delete from either end

Basic Data Structures I
- ADTs
- Lists, Stacks, Queues
- Sequential Allocation

Performance varies with implementation
- Stack: All access from one side
  - \( \text{push} \rightarrow \text{pop} \)
  - \( i \rightarrow \text{top} \rightarrow \text{null} \)
- Queue: FIFO list: enqueue inserts at tail and dequeue deletes from head
  - \( \text{enqueue} \rightarrow \text{dequeue} \)
**Cost model** (Actual cost)

- **Cheap**: No reallocation → 1 unit
- **Expensive**: Array of size \( n \) is reallocated to size \( 2n \)

**Dynamic (Sequential) Allocation**
- When we overflow, double

**Proof**:
- Break the full sequence after each reallocation → run
  \[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 \]
- At start of 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!

**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 \( \leq 5 \cdot m \)]

**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.
**Basic Data Structures III**

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

**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)
- $n \rightarrow n^2$ 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 → yes

**Dynamic Stack**:
- Showed doubling $\Rightarrow$ Amortized $O(1)$
- Other strategies?

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

**Multilists**: Lists of lists
- head $\rightarrow 0 \rightarrow a \rightarrow b \rightarrow 1 \rightarrow c \rightarrow d \rightarrow e \rightarrow$ tail

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

**Definitions**:
- **Depth**: path length from root
- **Height** (of tree): max. depth (max depth of any node)
- **Degree (of node)**: number of children
- **Degree (of tree)**: max. degree of any node

**Trees**: Basic Concepts and Definitions

**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**:
- Grandparent
- Parent
- Child
- Sibling
- Grandchild
- Leaf: no children
Representing rooted trees: Each node stores a (linked) list of its children

Node structure:

```
data -> nextSibling
  |   
  v     
firstChild
```

Wasted space?

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

E.g. n=5
nulls: 6

In Java:
```
class BTNode<E> {
  E data;
  BTNode<E> left;
  BTNode<E> right;
}
```

Trees Representation + Binary Trees (1)

(Not full) Full:

Full: Every non-leaf node has 2 children

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

called the Binary representation

Full: Every non-leaf node has 2 children
Traversals: How to (systematically) visit the nodes of a rooted tree?

Binary Tree Traversals (can be generalized)

Complete Binary Tree: All levels full (except last)

Preorder: process/visit v
Inorder: process/visit v
Postorder: process/visit v

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

Observation: Every extended binary tree is full

Eg. Inorder Threads:
Null left → inorder predecessor
Null right → "successor

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

Those wasteful null links...

Parent(i) = ⌈i/2⌉
Left(i) = 2i
Right(i) = 2i + 1

Another way to save space...

Threaded binary tree:
Store (useful) links in the null links. (Use a mark bit to distinguish link types.)
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 x:
- **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**: \( O(n) \)

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

Find:
- **How to find a key in the tree?**
  - Start at root \( p = \text{root} \)
  - if \( x < p.key \) search left
  - if \( x > p.key \) search right
  - if \( x == p.key \) found it!
  - if \( p == \text{null} \) not there!

Binary Search Trees
- **Basic definitions**
- **Finding keys**

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

Can we achieve \( O(\log n) \) time for all ops?

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

\[ \text{root} \rightarrow x \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow 11 \]

\[ 4 < x < 7 \]

\[ 1 < x < 2 \]

Value
\[
\text{find(Key x, BSTNode p)} =
\begin{cases}
    \text{null} & \text{if (p == null)} \\
    \text{null} & \text{if (x < p.key)} \\
    \text{null} & \text{if (x > p.key)} \\
    \text{null} & \text{if (p == null)} \\
    \text{return p.value} & \text{else return p.value}
\end{cases}
\]
Insert (Key x, Value v)
- find x in tree
- if found → error! duplicate key
- else: create new node where we "fell out"

Replacement Node?

Why did we do:

Why did we do:

p.left = insert(x, v, p.left) ?

p1: insert(14) → p1

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

p2 = new BSTNode

p2 = new BSTNode

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
}

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
3. x has two children
Find Replacement Node

```java
BSTNode findReplacement(BSTNode p) {
    if (p == null) return null; // Key not found
    else if (x < p.key) {
        p.left = findReplacement(p.left);
    } else if (x > p.key) {
        p.right = findReplacement(p.right);
    } else {
        BSTNode r = p.right;
        while (r.left != null) {
            r = r.left;
        }
        return r;
    }
    return p;
}
```

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

Binary Search Trees III
- deletion
- analysis
- Java

Example:
```
Before: 2 3 4 5 6 7 8
After: 2 3 4 5 6 7 11
```

Java Implementation:
- Parameterize key+value types: extends Comparable
  ```java
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 $O(n)$...
Java implementation (see notes for details)

```java
public class BS<T extends Comparable, V> {
    class Node {
        Key key;
        Value value;
        Node left, right;
        // ... constructor, toString...
    }

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

    private Node root;

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

- Inner class for node (protected)
- Local helpers (private or protected)
- Data (private)
- Public members (invoke helpers)
AVL 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 $h$)

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

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"

$n \geq \varphi^h = c \cdot \varphi^h \Rightarrow h \leq \log_\varphi n + c' \Rightarrow h \leq \log_\varphi n/\log_\varphi \varphi = O(\log n) \square$
AVL Node rebalance (AVL Node p)

Double rotations:
- Left-right (LR)
- Right-left (RL)

AVL Trees II
- Double rotations
- Insertion

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

How to rebalance? Bal = -2

Left-right heavy:
- Double rotations

AVL Node insert (Key k, Value v, AVL Node p)

Utilities:
- int height (AVL Node p)
- void updateHeight (AVL Node p)
- int balanceFactor (AVL Node p)
Cases: Balance factor -2
- Left-left heavy
  - Apply standard BST deletion
  - find key to delete
  - find replacement delete node
  - copy contents
  - delete replacement
  - rebalance

Left-right heavy
- Apply standard BST deletion
- find replacement delete node
- copy contents
- delete replacement
- rebalance

AVL Trees III
- Deletion
- Examples

AVL-Node delete (Key x, AVLNode p)
- same as BST delete
- return rebalance(p)

Examples:

Example 1:
- delete(7)

Example 2:
- insert(8)

Example 3:
- delete(7)

Example 4:
- delete(7)
  - LR rotate

Example 5:
- delete(7)
  - Rotate right
  - rotate LR

Example 6:
- delete(7)
  - Rotate right
  - rotate LR
Node types:
- 2-Node
  - 1 key
  - 2 children
  - Root: b
  - Subtrees: A, C
- 3-Node
  - 2 keys
  - 3 children
  - Root: b
  - Children: A, C, E

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 Ω(log n)

Roughly: \( \log_3 n \leq h \leq \log_2 n \)

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

Conceptual tool:
We’ll allow 1-nodes & 4-nodes temporary

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

Split: 4 → 2 + 2

Merge:
1 + 2 / 2 + 1 → 3

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
Insertion example:
- Insert 6

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

Delete Example:
- Delete 5

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

Example (continued)

Implementation:
```java
class TwoThreeNode {
    int children[3];
    Key key[2];
}
```
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 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.  

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?

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

Example:
- **2-3 Tree**:
- **Red-Black**:

EAYI.tw/newArynee.n.ndersoni99s/s

Root is black and has height \( O(\log n) \) that

\( \leq \)

nulls point to it → nil

"left-skewed" encoding

Corresponds to 2-3-4 trees
Restructuring Ops:
- **Skew**: Restore right skew
- If black node has red left child, rotate

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

**AA Tree**:

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 up) and move \( q \) up one level.

\[
\begin{align*}
\text{AA Node} & \text{ skew (AA Node p)} \\
& \begin{cases}
& \text{if (p == nil)} \text{ return p} \\
& \text{if (p.left.level == p.level)} \text{ right rotate p} \\
& \quad \text{AA Node } q = p\.left \\
& \quad p\.left = q\.right; q\.right = p \\
& \quad \text{return } q \leftarrow \text{new subtree root} \\
& \quad \text{else return } p \leftarrow \text{everything's fine}
\end{cases}
\end{align*}
\]

Red-Black + AA Trees II

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

\[
\begin{align*}
\text{AA Node split (AA Node p)} & \\
& \begin{cases}
& \text{if (p == nil)} \text{ return p} \\
& \text{if (p.right.right.level == p.level)} \leftarrows \text{ left rotation at p} \\
& \quad \text{AA Node } q = p\.right \\
& \quad p\.right = q\.left \\
& \quad q\.left = p \\
& \quad q\.level += 1 \leftarrow \text{move } q \text{ up a level} \\
& \quad \text{return } q \\
& \quad \text{else return } p \leftarrow \text{all okay}
\end{cases}
\end{align*}
\]
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(split(p))
```

**Red-Black and AA Trees III**

**Deletion:**
Two more helpers:

- **UpdateLevel:** If p's level exceeds \( l = 1 + \min(p.left.\text{level}, p.right.\text{level}) \), then set p's level to \( l \) and 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)
History:
1989: Seidel & Aragon
[Expllosion 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

Example: Insert: k, e, b, o, f, h, w (Std. BST) 1 2 3 4 5 6 7
Along any path - Insertion times increase

Treaps
- A tree that behaves as if keys are inserted in random order
- Worst case can be very bad O(n) height

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

Geometric Interpretation:
- key: x
- priority: y

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 and create a new leaf node.
- Assign random priority
- On backing out - check heap order and rotate to fix.

**Example:**
```
  3
 / \
13  45
 /   /  \
 b   f   c
```

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

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

**Proof:** Follows directly from BST analysis.

**Implementation:** (See pdf notes)

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

**Helpers:**
- return node of lowest priority $p$

**Restructure:**
- Performs rotation $p.left$ (if needed) to put lowest priority node at $p$.

**Example:**
```
  3
 /   /
14  45
 /   /  \
 k   e   t
```

**Example:**
```
  3
 /   /
14  45
 /   /  \
 k   e   t
```

**Treaps II**
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)

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

Idea: Add extra links to skip

Node Structure: (Variable sized)
```
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 -- ← 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 $\leq \frac{1}{n^{c-1}}$.

$\rightarrow$ Prob that any given node's level exceeds $k$ is $\frac{1}{2^k}$ [k consecutive heads]

$\rightarrow$ Prob that any of n node's level exceeds $k$ is $\leq \frac{n}{2^k}$ [n trials with prob $\frac{1}{2^k}$]

$\rightarrow$ Let $l = c \cdot \log n$ ($l = \log_2 n$)

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

$\leq \frac{n}{2^l} = \frac{n}{2^{\log_2 n}}$

$= \frac{n}{2^l} = \frac{n}{(2^{\log_2 n})^c}$

$= \frac{n}{n^c} = \frac{1}{n^{c-1}} \square$

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: 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: $E(i) = \text{Expt. num. nodes visited among top i levels}$.

Cases:

Cases (A)

Current node $\uparrow$ $\rightarrow$ same level $\downarrow$ From prior level

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

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

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

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

Basis: $E(0) = 0 \Rightarrow E(1) = 2 \cdot i$

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

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

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

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

$\rightarrow$ Let $n_i =$ no. of nodes that contrib. to level $i$.

$\rightarrow$ Prob that node at level $\geq i$ is $\frac{1}{2^i}$.

$\rightarrow$ Expected no. of nodes that contrib. to level $i = \frac{n}{2^i}$

$\Rightarrow E(n_i) = \frac{n}{2^i}$

Total space (expected) is:

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

$= n \sum_{i=0}^{l} \frac{1}{2^i} = 2n \square$
**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:
- 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:** find (75)

```plaintext
Delete (12)
```

```plaintext
Analysis: All operations run in O(log n) expected time due to randomness only - not sequential. ⇒ User cannot force poor performance.
```

```plaintext
Insert (24)
```

```plaintext
Note: Variation in running times
```

```plaintext
Find: (Similar to linked lists)
```

```plaintext
Key match found it!
```

```plaintext
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?
- 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

\[ \text{find}('a') \]

Tree restructures itself

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

Focus: Worst-case or randomized expected case

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

Splay Trees I

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

Final

Tree's height has reduced by ~ half!

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

still unbalanced!
ZigZig(p): [LL case]

Subtrees A, C move up↑

ZigZig(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) 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

Example:
splay(3)

Final
**Splay Trees III**

**Delete (x):**

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

**Dynamic Finger Theorem:**

- Keys: \( x_1, \ldots, x_n \). We perform accesses \( x_{i_1}, x_{i_2}, \ldots, x_{i_m} \).
- Let \( \Delta_j = i_j - i_{j-1} \), distance between consecutive items.
- **Thm:** Total access time is
  \[
  O(m + n \log n + \sum_{j=1}^{m} (1 + \log \Delta_j))
  \]

**Static Optimality:**

- Suppose key \( x_i \) is accessed with prob \( p_i \) (\( \sum p_i = 1 \)).
- Information Theory: Best possible binary search tree answers queries in expected time \( O(H) \) where
  \[
  H = \sum p_i \log \frac{1}{p_i} = \text{Entropy}
  \]

**Static Optimality Theorem:**

- Given a seq. of \( m \) ops. on splay tree with keys \( x_1, \ldots, x_n \), where
  - \( x_i \) is accessed \( q_i \) times. Let \( p_i = q_i / m \).
  - Then total time is
  \[
  O(m \sum p_i \log p_i)
  \]
**Multiway Search Trees:**

- Represent a set of elements using a tree structure.
- Nodes can have multiple children.

**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-Tree:**

- Perhaps the most widely used search tree.
- Databases.
- Numerous variants.

**B-Tree: of order m (≥3):**

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

**Node Structure:**

```c
class BTreeNode {
    int nChild; // no. of children
    BTreeNode child[M]; //children
    KeyKey key[M-1]; //keys
    ValueValue value[M-1]; //values
}
```

**Theorem:** A B-tree of order m with n keys has height at most \((\log n)/2\), where \(\gamma = \log(m/2).\)

(See full notes for proof.)

**Example:** m = 5

[Each node has: 3-5 children 2-4 keys]
**Key Rotation (Adoption)**
- A node has too few children $\lceil \frac{m}{2} \rceil - 1$
- Does either immediate sibling have extra? $\geq \lceil \frac{m}{2} \rceil + 1$
- 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 $m' = \lceil \frac{m}{2} \rceil$ and $m'' = m+1-\lceil \frac{m}{2} \rceil$

**Lemma:** For all $m \geq 2$, $\lceil \frac{m}{2} \rceil \leq m+1-\lceil \frac{m}{2} \rceil \leq m$ 

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

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

**B-Trees II**

**Node Merging:**
- A node has too few children $\lceil \frac{m}{2} \rceil - 1$
- Neither sibling has extra ($\geq \lceil \frac{m}{2} \rceil$)
- Merge with either sibling to produce node with $(\lceil \frac{m}{2} \rceil - 1) + \lceil \frac{m}{2} \rceil$ child

**Lemma:** For all $m \geq 2$, $\lceil \frac{m}{2} \rceil \leq \sqrt{m} - 1 \leq m$

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

Example: \( m = 5 \)

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

Example: \( m = 5 \)

B-Trees III
Geometric Search:
- Nearest neighbors
- Range searching
- Point Location
- Intersection Search

Multi-Dim vs. 1-dim Search?
Similarities:
- Tree structure
- Balance \(O(\log n)\)
- Internal nodes split
- External nodes data

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

Sofar: 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 with a region - cell
- Each internal node stores a splitter - subdivides the cell

Point: A \(d\)-vector in \(\mathbb{R}^d\)
\(p = (p_1, \ldots, p_d)\), \(p \in \mathbb{R}^d\)

Quadtrees & \(kd\)-Trees I

Representations:
- Scalars: Real numbers for coordinates, etc.
  \(\text{float} \) 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

\[
\begin{align*}
\text{float}[] & \quad \text{coord // coords} \\
\text{Point}(\text{int} d) & \quad \rightarrow \text{coord} = \text{new float}[d] \\
\text{int getDim()} & \rightarrow \text{coord.length} \\
\text{float get(int i)} & \rightarrow \text{coord[i]} \\
\ldots \text{others: equality, distance, toString...}
\end{align*}
\]
**Point Quadtree:**
- Each internal node stores a point.
- Cell is split by horizontal and vertical lines through point.

<table>
<thead>
<tr>
<th>(5, 4)</th>
<th>(4, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 2)</td>
<td>(7, 3)</td>
</tr>
</tbody>
</table>

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

**Histroy:** 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? $\rightarrow$ next

**kd-Tree:** Binary variant of quadtree
- Splitter: Horizontal or vertical line in 2-d (orthogonal plane owl)
- Cell: Still AABB
  - left: left/below
  - right: right/above

**Quadtrees & kd-Trees II**

**Find/pt Location:**
Given a query point $q$, is it in the tree, and if not which leaf cell contains it?
- Follow path from root down (generalizing BST find)

**Quadtrees - Analysis**
- Numerous variants!
  - PR, PMR, QR, QX, ... see Samet's book
- Popular in 2-d apps
  - (in 3-d, octrees)
- Don't scale to high dim
  - Out degree = $2d$
- What to do for higher dims?
**Example:**

![Kd-Tree Node](image)

```
class KDNode {
    Point pt // splitting point
    int cutDim // cutting coordinate
    KDNode left // low 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 $O(\log n)$.

**Value**

```java
public boolean onLeft(Point q) {
    return q[cutDim] < pt[cutDim];
}
```

```
public KDNode find(Point q, KDNode p) {
    if (p == null) return null;
    else if (q == p.pt) return p;
    else if (p.onLeft(q)) return find(q, p.left);
    else return find(q, p.right);
}
```
KDNode

Insertion:

```
if (p == null) // fell out?
    p = new KDNode(x, v, p, int 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
```

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)
```

```
2,3
    /
5,5

(2,3)
(3,4)

(5,5)
(4,1)
```

Analysis:

```
Run time: $O(h)$
```

Can we balance the tree?

- Rotation does not make sense

```
!! differ
```

```
rotate a
```

```
or?
```
Kd-Trees:
- Partition trees
- Orthogonal split
- Alternate cutting
  - dimension x, y, z, ...
- 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 Kth closest point
  - Find farthest point from q

This Lecture: O(√n) time alg.
  - for orthog. range counting queries
    in IR^2
  - General IR^d: O(n^{1-1/d})

Rectangle methods for kd-cells:
- Split a cell r by a split pt s ∈ r, along cutdim cd
  - high
  - left part
  - right part
  - Os[cd] + 1
  - pt in r

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]

Kd-Tree Queries

Axis-Aligned Rect in IR^d
- Defined by two pts:
  - low, high
    - high
    - low
  - Contains pt q ∈ IR^d iff
    - low_i ≤ q_i ≤ high_i

Useful methods:
- Let r, c - Rectangle
- q - Point
  - r.contains(q)
  - r.contains(c)
  - r.isDisjointFrom(c)
Orthogonal Range Query

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

Cases:
- p == null → fell out of tree → 0
- Query rect is disjoint from p's cell
  → return 0
  → no point of p contributes to answer
- Query rect contains p's cell
  → return p.size
  → every point of p's subtree contributes to answer
- Otherwise:
  - Rect + cell overlap → Recurse on both children

Kd-Tree Queries II

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.isDisjointFrom(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...}
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
  \( \Rightarrow \) \( O(n) \) time?
- At some we don't recurse at all!

Solving the Recurrence:
- Macho: Expand it
- Wimpy: Master Thm (CLRS)

Master Thm:
\[
T(n) = aT(\frac{n}{b}) + n^d + d \log_b a \\
\Rightarrow T(n) = n^{\log_b a}
\]

For us: \( a = 2 \)
\[
b = 4 \quad d = 0
\]
Since tree is balanced a child has half the pts + grandchild has quarter.

Recurrence: \( T(n) = 2 + 2T(\frac{n}{4}) \)

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
- \( R \)'s sides to 4 lines & analyze each one.
- Simpler: Extend \( R \) to 4

Kd-Tree Queries III

Stabbing: 3 cases
- cell is disjoint (easy)
- cell is contained (easy)
- cell partially overlaps or is stabbed by the query range (hard!)
Hashing: (Unordered) dictionary
- stores key-value pairs in array table [0..m-1]
- supports basic dict ops (insert, delete, find) in $O(1)$ expected time
- does not support ordered ops (getMin, findUp, ...)
- simple, practical, widely used

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

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 $\rightarrow$ randomize!
  - Let $H$ be a family of hash fns
  - Select $h \in H$ randomly
  - If $x \neq y$ then $\Pr(h(x)=h(y)) = \frac{1}{m}$
  - Let $p$ be large prime, $a,b \in \{0..p-1\}$ all random
  - $h_{ab}(x) = ((ax+b) \mod p) \mod m$

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

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

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

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 \((x,v) \rightarrow \text{table}[h(x)] = v\)
- Find \((x) \rightarrow \text{return table}[h(x)]\)
- Delete \((x) \rightarrow \text{table}[h(x)] = \text{null}\)

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

**Separate Chaining:**
- \(\text{table}[i]\) is head of linked list of keys that hash to \(i\).
- Example:
  - Keys \((x)\) vs \(h(x)\)
  - \(m = \#\) of keys
  - \(m = \) table size

**Hashing II**

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

**How to control \(\lambda\)?**
- Rehashing: If table is too dense/ too sparse, reallocate to new table of ideal size
- Designer: \(\lambda_{\min}, \lambda_{\max} \) - allowed \(\lambda\) values
  - \(\lambda_0 = \frac{\lambda_{\min} + \lambda_{\max}}{2} \) "ideal"
  - If \(\lambda < \lambda_{\min} \text{ or } \lambda > \lambda_{\max} \) ...

**Analysis:** Recall load factor
- \(\lambda = n/m\)
- \(n = \#\) of keys
- \(m = \) table size

**Proof:** On avg. each list has \(\lambda = \frac{n}{m}\)
- Success: 1 for head + half the list
- Unsuccess: \(\frac{1}{2} \times \) + all the list
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?

Linear Probing:
- $h(x), h(x)+1, h(x)+2, \ldots$
- 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!
  - $h(x)$ did not collide, but was not found
  - Table: $d \ w \ z \ p \ t \ c \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ldots$

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

Open Addressing:

Collision Resolution:
- May fail to find empty entry
  - (Try $m=4, j^2 \mod 4 = 0 \ or \ 1$ 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 $\lfloor m/2 \rfloor$ probe locations are distinct.

Pf: See latex notes.

Analysis:
- Improves secondary clustering
  - Hashing III

Analysis:

Linear Probing:
- $S_{LP} = \text{expected time for successful search}$
- $U_{LP} = \text{"unsuccessful"}$
- 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

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

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

Recap:
- Separate Chaining:
  Fastest but uses extra space (linked list)
- Open Addressing:
  - Linear probing: clustering
  - Quadratic probing:
- Probe sequences det'd by second hash fn. - g(x)

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 doub. hash. if successful
- \( U_{DH} \) = Exp. if unsuccessful
Recall: Load factor \( \lambda = n/m \)

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

Problem:
- insert(a):
  \( h(x) \) [found]: "a" found!
- delete(o):
  \( h(x) \) [not found!]
- find(a):
  \( h(a) \) [not found!]

Find(x):
- Visit entries on probe sequence until:
  - found \( x \) \( \Rightarrow \) return \( v \)
  - hit empty \( \Rightarrow \) return null
  \( \Rightarrow \) Not found!

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

- Proof is nontrivial (skip)

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!)

Dictionary Operations:
**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

**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 $\leq \log_{3/2} n \approx 1.71 \log n$

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

**Example**:

```
Example:
```

```
How to rebuild $p$:
- inorder traverse $p$'s subtree $\rightarrow$ 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($x, L, R$)
```

```
Final
```

```
Time = $O(k)$
```

```
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 $\leq \log_{3/2} n \approx 1.71 \log n$
```
**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, $size(p) =$ no. of nodes in $p$'s subtree
    - if $size(p.child) > \frac{2}{3} size(p)$
      - $p = \text{rebuild}(p)$
      - break

**Delete:**
- Same as std BST
- $n = \frac{m}{2}$
- if $m > 2n$, rebuild(root)

**Scapegoat Trees**

**Example:**
- Insert 5

**Proof:** By contradiction
- Suppose $p$'s depth $> \log_{3/2} n$
- but $\forall$ ancestors
- $\exists$ node $p$ of depth $> \log_{3/2} n$, then $\exists$ ancestor of $p$ that satisfies scapegoat condition

**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

**How to compute $size(p)$?**
- Can compute it on the fly
- While backing out, traverse "other sibling"
- Too slow? No!
  → Charge to rebuild.
**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
Range Tree Applications:
- Range trees can be applied to a variety of query problems.

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

Minimization/Maximization - 3-Sided Min Query
Given a set $P$ of $n$ pts in $\mathbb{R}^2$, a query consists of x-interval $[x_0, x_1]$ and $y$ value $y_0$. Return the lowest pt in 3-sided region $x_0 \leq x \leq x_1$, $y \geq y_0$

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

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

Return a count of the number of pts of $P$ lying within the triangle.
### 3-Sided Min Query

Return lowest in region $x_0 \leq x \leq x_1$, $y \geq y_0$

![Diagram showing the region and answer](image)

#### Data Structure:
- Build a range tree for $x$.
- Aux. trees are range trees for $y$ that support find larger.

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

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

### Skewed rectangle query:

Transform coordinates to make orthog range query

- $q^{-} = (x^{-}, y^{-})$
- $q^{+} = (x^{+}, y^{+})$

#### Line equation:

\[ y = x + (q_{y} - q_{x}) \]

- $q_{x} - q_{x} \leq p_{y} - p_{x} \leq q_{y} - q_{x}$

#### Map each $p = (p_{x}, p_{y}) \in \mathbb{P}$

- $p' = (p_{x}', p_{y}') = (p_{x}, p_{y} - p_{x})$

Let $\mathbb{P}'$ be resulting set.

#### Build std. range tree for $\mathbb{P}'$

- Return ans. to query

- $q_{x} - q_{x} \leq x \leq q_{x}^{+}$
- $q_{y} - q_{x} \leq y \leq q_{y}^{+} - q_{x}$
**NE Right Triangle Query**

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

**Build a 3D range tree on $P'$**

NE triangle query becomes:

- $q_x \leq x \leq q_x + l$
- $q_y \leq y \leq q_y + l$
- $q_x + q_y \leq z \leq q_x + q_y + l$

**Space:** $O(n \log^2 n)$

**Query time:** $O(\log^3 n)$
Can we do better?

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: $O(n)$ time
  - Reporting: $O(k \log n)$ time

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 $n$

Range trees are more limited

Layering: Combining search structures

- Suppose you want to answer a composite query with multiple criteria:
  - Medical data: Count subjects
    - Age range: $a_{i_0} \leq \text{age} \leq a_{i_n}$
    - Weight range: $w_{i_0} \leq \text{weight} \leq w_{i_n}$
  - Design a data structure for each criterion individually
  - Layer these structures together to answer full query

→ Multi-Layer Data Structures

Call this a 1-Dim Range Tree:

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 $O(k + \log n)$)

1-Dim Range Tree:

- Goal: Express answer as disjoint union of subsets
- Method: Search for $Q_{i_0} + Q_{i_n}$
  - Take maxima of subtrees

Layering:

Combining search structures

Approach:

- Balanced BST (e.g. AVL, RB, ...)
- Assume extended tree
- Each node $p$ stores no. of entries in subtree: $p.size$
Recursive helper:

```c
int range1Dx(Node p, Intv Q=\([Q_L, Q_R]\), Intv C=\([x_0, x_1]\))
```

initial call:
```c
range1Dx(root, Q, C)
```

More details:

Given a 1D range tree T:

- Let \( Q=\([Q_L, Q_R]\) \) 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=\([-\infty, +\infty]\) \)

Cases:

\( p \) is external:
- if \( p.pt.x \in Q \) → 1 else → 0

\( p \) is internal:
- \( C \subseteq Q \) ⇒ 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

Analysis:

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

Thm: Given 1D 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 range 1Dx to find all subtrees that contribute
  - For each such node \( p \)
    - run range 1Dy on \( p.aux \)
  - Return sum of all result

\[ \text{x-range tree} \]

\[ \text{p.aux} \]

\[ \text{y-range tree} \]

\[ \text{qhi} \]

\[ \text{qlo} \]

\[ \text{Q_{lo,x}} \]

\[ \text{S(p)} \]

\[ \text{Q_{hi,x}} \]

\[ \text{Q_{lo,y}} \]

\[ \text{Q_{hi,y}} \]

\[ \text{int range2D(Node p, Rect Q, Inv, C=[x_0, x_1])} \]

\[ \text{if (p is external) return } p.p.t \in Q ? \frac{1}{2} \]

\[ \text{else if (Q \times \text{contains } C)} \]

\[ \text{C \in Q\text{'s } x\text{-projection}} \]

\[ [y_0, y_1] = [-\infty, +\infty] \]

\[ \text{// init y-cell} \]

\[ \text{return range1Dy(p.aux, Q, [y_0, y_1])} \]

\[ \text{else if (Q.x is disjoint of C) return 0 } \]

\[ \text{else} \]

\[ \text{// partial x-overlap} \]

\[ \text{return range2D(p.left, Q, [x_0, p.x])} \]

\[ + \text{ range2D(p.right, Q, [p.x, x_1])} \]

\[ \text{int} \]

\[ \text{Range Trees III} \]

\[ \text{Higher Dimensions?} \]

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

Analysis: The 1D x search takes
of \( O(\log n) \) time & generates
\( O(\log n) \) calls to 1Dy search
\Rightarrow Total: \( O(\log n \cdot \log n) = O(\log^2 n) \)

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 \( y \).
**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}

**Example**: $\Sigma' = \{a, b, c, \ldots\}$
- Eq. $\Sigma' = \{0, 1\}$  Let $k = |\Sigma'|$
- Assume chars coded as ints: $a=0, b=1, \ldots, z=k-1$

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

**Analysis**: Smaller by factor $k$
- Search Time: Larger by factor of $k$

**Example**:

**Tries and Digital Search Trees I**

**How to save space?**
- de la Briandais trees
  - Store 1 char. per node
  - $x = x \Rightarrow$ try next char in $\Sigma$
  - $x = x \Rightarrow$ advance to next character of search string

**Analysis**:
- Space: Smaller by factor $k$
- Search Time: Larger by factor of $k$
Patricia Tries:
- Improves trie by compressing degenerate paths
- PATRICIA = Practical Alg. to Retrieve Info. Coded in Alpha
- Late 1960’s: Morrison & Guchenhberger
- Each node has index field, indicates which char to check next (Increase with depth)

Default: Substring identifier for $S_i$ is shortest prefix of $S_i$ unique to this string $S_1$ - $AMA$ 
Eq: $ID(S_1) = "ama\$"

Example: $S = \{0, 1, 2, 3, 4, 5, 6\}$

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

Notation: $S = a_1a_2a_3...a_m\$
- Suffix: $S_i = a_ia_{i+1}...a_m\$
  - Special terminal

$Q$: What is minimum substring needed to identify suffix $S_i$?

Tries and Digital Search Trees II

- Example:
  - ID
  - $S_0$: ajam...
  - $S_1$: aj
  - $S_2$: pajam...
  - $S_3$: apaj...
  - $S_4$: mapaj...
  - $S_5$: amapaj...
  - $S_6$: amapa...

Dealing with long Paths:
- To get both good space & query time efficiency, need to avoid long, degenerate paths.
- Path compression!

Example:
- $ID$: $S_6$: ajama$\$
- $S_0$: amapa$\$
- $S_1$: amapaj$\$
- $S_2$: mapaj$\$
- $S_3$: apaj$\$
- $S_4$: pajam$\$
- $S_5$: ajam$\$
- $S_0$: aj

Same data structure – Drawn differently
- Just easier to read the strings out...
- Same data struc.
- $K$: (No. of nodes) + (Storage for strings)
- $Q$: What is minimum substring needed to identify suffix $S_i$?

Analysis:
- Query time: (Same as std trie) $\sim$ search string length (may be less)
- Space:
  - No. nodes $\sim$ No. of strings (irresp. of length)

Example:
- $S = \{0, 1, 2, 3, 4, 5, 6\}$
- $ID(S_0) = "aj",
- $ID(S_1) = "aj\$
- $ID(S_2) = "ama\$"
- $ID(S_3) = "ama\$
- $ID(S_4) = "ama\$
- $ID(S_5) = "ama\$
- $ID(S_6) = "ama\$

Same data structure – Drawn differently
- abcede
- Discriminator
- Skipped substring
- $cc$
- terminal
**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

**Substring Queries:**

- How many occurrences of \( t \) in \( S \)?

  - Search for target string \( t \) in trie
  - if we end in internal node (or midway on edge) - return
  - else (fall of on external node)
    - compare target with string
    - if matches - found 1 occurrence
    - else - no occurrences

**Example:**

Search ("ama") \( \rightarrow \) End at intern node

Search ("amapaj") \( \rightarrow \) End at extern node

- Space: \( O(n) \) nodes
  - \( O(n \cdot k) \) total space
  - \( (k = |\Sigma| = O(1)) \)

- Search time: \( n \) to length of target string

- Construction time: \( -O(n \cdot k) \) [non-trivial]

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

**Geometric Applications:**

**PR k-d Tree:** k-d tree based on midpoint subdivision

Assume points lie in unit square

- How many occurrences of \( t \) in \( S \)?

  - Search for target string \( t \) in trie
  - if we end in internal node (or midway on edge) - return
  - else (fall of on external node)
    - compare target with string
    - if matches - found 1 occurrence
    - else - no occurrences

**Example:**

Search ("ama") \( \rightarrow \) End at intern node

Search ("amapaj") \( \rightarrow \) End at extern node

- Space: \( O(n) \) nodes
  - \( O(n \cdot k) \) total space
  - \( (k = |\Sigma| = O(1)) \)

- Search time: \( n \) to length of target string

- Construction time: \( -O(n \cdot k) \) [non-trivial]
Binary Encoding:
- Assume our points are scaled to lie in unit square $0 \leq x, y < 1$ (can always be done)
- Represent each coordinate as binary fraction:
  $x = 0.a_1a_2a_3 \ldots$, $a_i \in \{0, 1\}$
  $y = \sum a_i \cdot 2^{-i}$

Example:

How do we extend to 2-D?

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 $\Sigma^* = \{0, 1\}$)
- 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
  $x = 0.a_1a_2a_3 \ldots$
  $y = 0.b_1b_2b_3 \ldots$
  $z = 0.c_1c_2 \ldots$

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 $\Phi(P) = \{\phi(p_1), \phi(p_2), \ldots, \phi(p_n)\}$
(in binary strings)
Then the PR kd-tree for $P$ is equivalent to binary trie for $\Phi(P)$.

Bit Interleaving:
Given a point $p = (x, y)$
$0 \leq x, y < 1$
let $x = 0.a_1a_2a_3 \ldots$ in binary
$y = 0.b_1b_2b_3 \ldots$

Define:
$\phi(x, y) = a_1b_1a_2b_2a_3b_3 \ldots$
Called Morton Code of $p$

How do we extend to 2-D?

How do we extend to 2-D?

How do we extend to 2-D?

How do we extend to 2-D?

How do we extend to 2-D?

Proof: By induction on no. of bits
Let $x = 0.a_1a_2a_3 \ldots$ $y = 0.b_1b_2b_3 \ldots$
and consider just $\phi(x, y) = a_1b_1a_2b_2a_3b_3 \ldots$

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