Data structures are **FUNDAMENTAL**!

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

Basic elements in study of data structures

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

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

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

Common:

- $O(1)$: constant time [Hash map]
- $O(\log n)$: log time (very good!) [Binary search]
- $O(n^p)$: ($p$ is constant) Poly time
eq $O(n)$ [Geometric search]

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

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

Asymptotic Analysis:
- Run time as a function of $n$ ← no. of items
- Worst-case, average-case, randomized
- Amortized: Average over a series of ops.
Linear List ADT:
Stores a sequence of elements \( \langle a_1, a_2, \ldots, a_n \rangle \). 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 element
- \( \text{delete}(i) \) - deletes \( i \)th item

Implementations:
- Sequential: Store items in an array
  \[
  a_1 \ a_2 \ a_3 \ \ldots \ a_n
  \]
- Linked allocation: linked list
  - Singly: \( \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_i \rightarrow \text{null} \)
  - Doubly: \( \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_i \rightarrow \text{null} \)

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

Stack: All access from one side
- \( \text{push} \rightarrow \text{pop} \)

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

Performance varies with implementation
**Cost model (Actual cost)**

**Cheap:** No reallocation \(\rightarrow 1\) unit

**Expensive:** Array of size \(n \Rightarrow 2n+1\)

\[ \text{is reallocated to size } 2n \]

**Dynamic (Sequential) Allocation**

- When we overflow, double

**Basic Data Structures II**

- Amortized analysis of dynamic stack

**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 \(\rightarrow \text{run} \)
- 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
  \[ \Rightarrow 1 \text{ for each op} \]
  \[ \Rightarrow 4 \text{ for deposit} \]
- Next reallocation costs \(4n\), but we have enough saved!

**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 \(\leq 5m\)]
**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)
\[ n \rightarrow n^2 \text{ or } n \rightarrow \sqrt[3]{n} \]

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

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

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

**Multilists**: Lists of lists

**Sparse Matrices**:
- An \( n \times m \) matrix has \( n \cdot m \) entries and takes (naively) \( O(n \cdot m) \) space
- Sparse matrix: Most entries are zero

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

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

**Leave as exercise (Spoiler alert!)**
- Fixed increment \( \rightarrow \) no
- Fixed factor \( \rightarrow \) yes
- Squaring \( \rightarrow \) ?? (depends on cost model)
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 of non-root nodes

Degree (of node): number of children

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

Trees: Basic Concepts and Definitions

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

"Family" Relations

- grand parent
- parent
- child
- siblings
- grandchild
- leaf: no children
Representing rooted trees:
Each node stores a (linked) list of its children

Node structure:

Trees Representation + Binary Trees (I)

(Not full) Full:

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

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

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

root:

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

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

Binary Tree Traversals (can be generalized)
- Preorder: root ...? v
  - process/visit v
  - traverse T_L recursive
  - traverse T_R

Complete Binary Tree: All levels full (except last)

Challenge: Non-recursive traversals

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

Those wasteful null links...

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

Recursive Pre/post/traversal of a binary tree:
- Preorder: visit root first, then children
- Inorder: visit left children first, then root, then right children
- Postorder: visit right children first, then root, then left children

Another way to save space:

In order: 
- Visit the root first, then visit all the left children, then the right child
- Visit the right children first, then visit all the left children

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

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

In order threads:
- Null left -> inorder predecessor
- Null right -> "successor
Dictionary:
- **Insert** \((\text{Key } x, \text{Value } v)\)
  - Insert \((x, v)\) in dict. (No duplicates)
- **Delete** \((\text{Key } x)\)
  - Delete \(x\) from dict. (Error if \(x\) not there)
- **Find** \((\text{Key } x)\)
  - 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 \(v\):
  - Store for quick access & updates
  - Ordered: Assume that keys are totally ordered: \(<, \geq, \leq\)

Efficiency:
- Depends on tree's height
  - **Balanced**: \(O\left(\log n\right)\)
  - **Unbalanced**: \(O(n)\)

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

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

Binary Search Trees
- Basic definitions
- Finding keys

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

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

Value: `find(\text{Key } x, \text{BSTNode } p)`
- if \((p = \text{null})\) return null
- else if \((x < p.\text{key})\)
  - return \(\text{find}(x, p.\text{left})\)
- else if \((x > p.\text{key})\)
  - return \(\text{find}(x, p.\text{right})\)
- else return \(p.\text{value}\)`
**Insert (Key x, Value v)**
- find x in tree
- if found ⇒ error! duplicate key
- else: create new node where we "fell out"

**Replacement Node?**

**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
3. x has two children
BSTNode delete (Key x, BSTNode p)
if (p == null) return null
else
  if (x < p.key) 
    p.left = delete (x, p.left) 
  else if (x > p.key)
    p.right = delete (x, p.right) 
  else if (either p.left or p.right null)
    if (p.left == null) 
      return p.right 
    if (p.right == null) 
      return p.left 
  else
    r = find replacement (p) 
    copy r's contents to p 
    p.right = delete(r.key, p.right) 
return p

Example:
Del (5)
Java implementation (see notes for details)

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

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

    // constructor, toString...
}

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

private Node root;

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

Inner class for node (protected)

Local helpers (private or protected)

Data (private)

Public members (invoke helpers)
Balance factor:

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

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.

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

Worst cases:

- height:
  \[ h = 0, 1, 2, \ldots, h \]

- nodes:
  \[ n = 1, 2, 4, 7, 12, 20, \ldots \]

- \(n_1 = 2, 3, 5, 8, 13, 21, \ldots\)

Recall: \(F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}\)

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

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

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

- \(h = 0: n(h) = F_1 = 1\)
- \(h = 1: n(h) = 2 = F_2\)
- \(h \geq 2: n(h) = 1 + n(h-1) + n(h-2) = 1 + (F_{h-2} + F_{h-1}) = F_{h+1} - 1\)

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

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

- \(n \geq F_h = c \cdot \varphi^h \Rightarrow h \leq \log_\varphi n + c\)

- \(h \leq \log_2 n / \log_2 \varphi = O(\log n)\)
AVL Trees II
- double rotations
- insertion

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

Utilities:
- int height(AVLNode p) return { p == null → -1
  or. → p.height
- void updateHeight(AVLNode p) p.height = 1 + max(height(p.left), height(p.right))
- int balanceFactor(AVLNode p) return height(p.right) - height(p.left)

AVLNode rebalance(AVLNode p)
if (p == null) return p
if (balanceFactor(p) < -1)
  if (ht(p.left.left) ≥ ht(p.left.right))
    p = rotateRight(p)
  else p = rotateLeftRight(p)
else if (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)

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

AVL Tree: simpler than balance factor

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

How to rebalance? Bal = -2

Left-right heavy:

Right-left heavy:

Double rotations: left-right

right-left to Double rotations:
Cases: Balance factor: -2
- Left-Left heavy
  - Apply standard BST deletion
  - Find key to delete
  - Find replacement node
  - Copy contents
  - Delete replacement
  - Rebalance

Deletion: Basic plan
- Apply standard BST deletion
- Find key to delete
- Find replacement node
- Copy contents
- Delete replacement
- Rebalance

Example 1:
- AVL Trees III
- Deletion
- Examples

AVL Node delete (Key x, AVL Node p)
- Same as BST delete
- Return rebalance(p)

Examples:
- Example 1
- Example 2
- Example 3
- Example 4

Notes:
- Examples illustrate deletion and insertion operations in AVL trees.
Node types:
- 2-Node: 1 key, 2 children
- 3-Node: 2 keys, 3 children

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

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

Example:
- 2-3 tree of height 2

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 $O(\log n)$

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

Identical heights

2-3 Trees

Split: 4 $\rightarrow$ 2+2

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

Insert into parent

Steal b from parent

Split

**Insertion example:**
- Insert 6:
  - New root: 8
  - Split: 8
- Insert 8:
  - New root with 10 children

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

**Implementation:**
```java
class TwoThreeNode {
    int children[];
    int key[];
}
```

**Example (continued):**
- Example:
  - Merge
  - Adopt
- Deletion remedy:
  - Have a 3-node neighboring sibling → adopt
  - O.w.: Merge with either sibling + steal key from parent
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**

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)

**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?

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? $p.\text{left. level} == p.\text{level}$

Split: If a black node has a right-right red chain, do a left rotation at $p$ (bringing its right child up) and move $q$ up one level.
  - How to test? $p.\text{level} == p.\text{right. level} == p.\text{right.right. level}$ (not needed, levels are monotone)

Example:
- 2-3 Tree:
  - AA tree:
  - AA Insertion: - Find the leaf (as usual)
    - Create new red node
    - Back out applying skew + split

AA Node $\text{split}(\text{AA Node } p)$
  - if ($p == \text{nil}$) return $p$
  - if ($p.\text{right.right. level} == p.\text{level}$)
    - AA Node $q = p.\text{right}$
    - $p.\text{right} = q.\text{left}$
    - $q.\text{left} = p$
    - $q.\text{level} = 1$
    - move $q$ up a level
  - else return $p$ (all okay)

AA Node $\text{skew}(\text{AA Node } p)$
  - if ($p == \text{nil}$) return $p$
  - if ($p.\text{left. level} == p.\text{level}$)
    - Right rotate $p$
    - AA Node $q = p.\text{left}$
    - $p.\text{left} = q.\text{right}$
    - $q.\text{right} = p$
    - return $q$ (new subtree root)
  - else return $p$ (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 III

**Deletion:**
Two more helpers:

**Update Level:** 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 After Delete (p):**
- update p's level
- skew (p), skew(p.right)
split(p), split(p.right)
deletion: Same as AVL deletion, but end with: return fix AfterDelete (p)
**History:**
1989: Seidel & Aragon
[Explosion of randomized algorithms]
Later discovered this was already known: Priority
Search Trees from different context (geometry)
McCreight 1980

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

**Example:** Insert: k, e, b, o, f, h, w
(Std. BST)

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

**Geometric Interpretation:**

- **Obs:** In a standard BST, keys are by inorder + insert times are in heap order (parent < child)
- **Treap:** Each node stores a key + a random priority
  Keys are in inorder
  Priorities are in heap order

- **Example:**
  - 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 + 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 $p$.

**Deletion:** (cute solution) Find node to delete. Set its priority to $+\infty$.
Rotate it down to leaf level + unlink.
Ideal Skip List:
- Organize list in levels
  - Level 0: Everything
  - Level 1: Every other
  - Level 2: Every fourth
  - Level i: Every $2^i$

Sorted linked lists:
- Easy to code
- Easy to insert/delete
- Slow to search... $\mathcal{O}(n)$

Idea: Add extra links to skip

How to generalize?

Example:

Node Structure:
```java
class SkipNode{
    Key key
    Value value
    SkipNode[] next
}
```

Value `find(Key x)`:
```java
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 $\leq \frac{1}{n}$. 

\[ \text{Prob that any given node's level exceeds } l \text{ is } \frac{1}{2^l} \]  

\[ \text{Prob that any of } n \text{ node's levels exceeds } l \text{ is } \leq \frac{n}{2^l} \]  

\[ \text{Let } l = c \cdot \log n \Rightarrow \text{Prob that max level exceeds } c \cdot \log n \text{ is:} \]  

\[ \leq \frac{n}{2^l} = \frac{n}{2^{c \cdot \log n}} = n / (2^{\log n}^c) = n/n^c = \frac{1}{n^{c-1}} \]

**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:

\[ 2^0 + 1 + 2^1 + 2 + 1 + 3 = 9 \]

- 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:

\[ E(\sum_{i=0}^{\infty} n_i) = \sum_{i=0}^{\infty} E(n_i) = \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n \]

**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.

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

**Cases:**

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

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

**Basis:** $E(0) = 0 \Rightarrow E(i) = 2 + E(i-1)$

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

\[ \Rightarrow \text{We visit 2 nodes per level on average.} \]
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:
- 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

Insert (24)
- visit, don't save
- visit, save reference

Delete (12)
- visit, don't save
- visit, save reference

Analysis: All operations run in time \( \sim \text{find} \Rightarrow O(\log n) \) expected

Note: Variation in running times due to randomness only - not sequenced
⇒ 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 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
\[ \text{find}("a") \]
UGH!
- Tree restructures itself

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

Focus: Worst-case or randomized expected case

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)

Idea II: Rotate 2 at a time - upper + lower
...final result: c
still unbalanced!
**Splay Trees II**

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

**Find (x):**
- root ← splay (x)
  - if (root.key == x) return root.value
  - else return null

**Example:**
- splay (3)

**Zig (p):** [L case]
- p: [Subtrees A, C move up]

**ZigZig (p):** [LL case]
- p: [Subtrees A, C move up]

**ZigZaa (p):** [LR case]
- p: [Subtrees C, E of p move up]

**ZigZag (p):** [L case]
- p: [Subtree A moves up, C unchanged]
**Dynamic Finger Theorem:**

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

**Thm:** Total access time is

$$O(m + n \log n + \sum_{i=1}^m (1 + \log \Delta_j))$$

---

**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
  
  $H = \sum p_i \log \frac{1}{p_i}$ = Entropy

---

**Static Optimization 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

$$O(m \sum p_i \log \frac{1}{p_i})$$
Multiway Search Trees:

- Most large data structures reside on disk storage
- Organized in blocks or pages
- Latency: High start-up time
- Want to minimize number of blocks accessed

**B-Tree**

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

**B-Tree of order m (≥3)**

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

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

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

**Node Structure**:

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

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

(See full notes for proof)
**Key Rotation (Adoption)**
- A node has too few children \( \left\lceil \frac{m}{2} \right\rceil - 1 \)
- Does either immediate sibling have extra? \( \geq \left\lceil \frac{m}{2} \right\rceil + 1 \)
- Adopt child from sibling & rotate keys
- When applicable - preferred

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

**Node Splitting:**
- After insertion, a node has too many children \( m + 1 \)
- We split into two nodes of sizes \( m' = \left\lceil \frac{m}{2} \right\rceil \) and \( m'' = m + 1 - \left\lceil \frac{m}{2} \right\rceil \)

**Lemma:** For all \( m \geq 2 \),
\[ \left\lceil \frac{m}{2} \right\rceil \leq m + 1 - \left\lceil \frac{m}{2} \right\rceil \leq m \]
\[ \Rightarrow m' + m'' \text{ are valid node sizes} \]

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

**B-Trees II**

**Lemma:** For all \( m \geq 2 \),
\[ \left\lceil \frac{m}{2} \right\rceil \leq 2 \left\lceil \frac{m}{2} \right\rceil - 1 \leq m \]
\[ \Rightarrow \text{Resulting node is valid} \]
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
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:

```
p: b
  a c
  d e
  f
```

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!
  - Maintain $n, m < 0$

Find:
- Same as std. BST
- Tree height $\leq \log_{\frac{1}{3}} n \approx 1.71 \log n$

Delete:
- $n-- \rightarrow$ If $m > 2n$ rebuild

How to rebuild?

1. Inorder traverse $p$'s subtree → array $A[]$
2. buildSubtree($A[]$)

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

Time = $O(k)$
Details of Operations:

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

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

**Scapegoat Trees**

- **Must there be a scapegoat?** Yes!

**Proof**: By contradiction

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

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

- \( d \geq \log_{3/2} n \)
  - \( \text{size}(u\text{'s child}) \leq \frac{2}{3} \cdot \text{size}(u) \)
  - \( \text{size}(u) \leq \frac{2}{3} \cdot \text{size}(u) \)
  - \( 1 \text{ node}: \text{size}(p) \leq \frac{2}{3} n \)
  - \( \Rightarrow \frac{2}{3} d \leq n \)
  - \( d \geq \log_{3/2} n \)
Scapegoat Trees

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 $\approx$ number of nodes in tree
  $\rightarrow$ Amortize rebuild time against delete ops
- **Insert:** Based on potential argument
  $\rightarrow$ It takes $\approx k$ ops to cause a subtree to size $k$ to be unbalanced.
  $\rightarrow$ Charge rebuild time to these operations
Geometric search:
- So far: 1-dimensional keys

Partition Trees:
- Tree structure based on:
  - Internal nodes: split
  - External nodes: data

Similarities:
- Balance (Log n)
- Internal node split
- External nodes data

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

Multi-Dim vs. 1-Dim Search?
- Point Location
- Intersection Search
- Range searching
- Nearest neighbors

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

- Point: A vector in \( \mathbb{R}^d \)
- External nodes store points
- Each internal node stores a splitter - subdivides the cell
- Built from these:
  - Class Point
  - float[Dim] coord[]
  - get(int i) = coord[i]
  - set(int i, float f)
  - float length
  - Other: equality distance

- Scalar: Real numbers
- Built from these:
  - Built from:
- Other geom objects

- Quadtrees & Kd-Trees
- Quadtree rotation may not be meaningful
Point Quadtree:
- Each internal node stores a point
- Cell is split by horiz. + vertic. lines through point

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

Quadtrees & kd-Trees

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

History: Bentley 1975
- Called it 2-d tree (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

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

Kd-Tree Node:
```
class KDNode {
    // splitting point
    Point pt
    // cutting coordinate
    int cutDim
    // left side
    KDNode left
    // right side
    KDNode right
}
```

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

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

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

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

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

**Kd-Tree Insertion:**
(Similar to std. BSTs)
- Descend tree until cutting. → find pt → Error! duplicate
- fall out <→ (Although we draw extended trees, lets assume standard trees)
- create new node
- set cutting dim

**Quadtrees & kd-Trees**

**Example:**
```
Example:
```

**Analysis:**
- Runtime: \( O(h) \)

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

**Deletion:**
- Descend path to leaf
- If found:
  - leaf node → just remove
  - internal node → find replacement → copy here → recur. delete replacement
  - Rebalance by Rebuilding:
    - Rebuild subtrees as with scapegoat trees
    - \( O(\log n) \) amortized
    - Find: \( O(\log n) \) guaranteed

This is the hardest part. See Latex notes.

Tree height
Kd-Trees:
- Partition trees
- Orthogonal split
- Alternate cutting dimension $x, y, x, y,...$
- Cells are axis-aligned rectangles (AABB)

Queries:
- Orthogonal range queries
  - Given query rect. (AABB) count/report pts in this rect.
  - Other range queries?
    - Circular disks
    - Halfplane
- Nearest neighbor queries
  - Given query pt, return closest pt in the set
  - Find K-th closest point
  - Find farthest point from q

This Lecture: $O(\sqrt{n})$ time alg. for orthog. range counting queries in $\mathbb{R}^2$
$\rightarrow$ General $\mathbb{R}^d$: $O(n^{1-1/d})$

Kd-Tree Queries I

Rectangle methods for kd-cells:
- Split a cell $r$ by a split pt $s \in r$, along cut dim $d$.
  - $r_{\text{high}}$: high $[cd] \leftarrow s[cd]$.
  - $r_{\text{low}}$: low $= r_{\text{low}}$ but low $[cd] \leftarrow s[cd]$
  - $r_{\text{leftPart}}(cd, s)$ returns rect with $\text{low} = r_{\text{low}}$
    $\rightarrow$ high $= r_{\text{high}}$ but high $[cd] \leftarrow s[cd]$
  - $r_{\text{rightPart}}(cd, s)$

Axis-Aligned Rect in $\mathbb{R}^d$
- Defined by two pts: low, high
- Contains pt $q \in \mathbb{R}^d$ iff $\text{low}_i \leq q_i \leq \text{high}_i, 1 \leq i \leq d$

Useful methods:
- Let $r, c$ - Rectangle
- $q$ - Point
  - $r_{\text{contains}}(q)$
  - $r_{\text{contains}}(c)$
  - $r_{\text{isDisjointFrom}}(c)$
Orthog. Range Query
- Assume: Each node p stores:
  - p.pt: splitting point
  - p.cutDim: cutting dim
  - p.size: no. of pts in p's subtree
- Tree stores ptr. to root and bounding box for all pts.
- Recursive helper stores current node p + p's cell.

Cases:
- p == null → fell out of tree → 0
- Query rect is disjoint from p's cell → return 0
- Query rect contains p's cell → return p.size
- Query rect overlaps 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 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 ++ // pt in range
        ct += rangeCount(R, p.left, cell.leftPart(p.cutDim, p.pt))
        ct += rangeCount(R, p.right, cell.rightPart...)
    } return ct;
```

Final answer = 1+1+1+2 = 5
**Theorem:** Given a balanced $Kd$-tree storing $n$ pts in $\mathbb{R}^2$, using alternating cut dim, orthog. range queries can be answered in $O(n \log n)$ 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:**
$$T(n) = aT(\frac{n}{b}) + n^d + d\log_b a$$
$$\Rightarrow T(n) = n^{\log_b a}$$

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

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 $O(n \log n)$ cells can be stabbed by $l$.

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

**Cases:**
- $p$ splits horizontally
- $p$ splits vertically
- $p$ splits both

Simpler: Extend $R$'s sides to 4 lines & analyze each one.
Hashing: (Unordered) dictionary
- stores key-value pairs in array table [0..m-1]
- supports basic dict. ops. (insert, delete, find) in O(1) expected time
- does not support ordered ops (getMin, findUp, ...)
- simple, practical, widely used

Recap: So far, ordered dicts.
- insert, delete, find
- Comparison-based: <, ==, >
- getMin, getMax, getK, findUp...
- Query/Update time: O(log n)
  → Worst-case, amortized, random.
  ♻ Can we do better? O(1)?

Universal Hashing:
Even better → randomize!
- Let H be a family of hash fns
- Select h ∈ H randomly
- If x ≠ y then Prob(h(x) = h(y)) = \frac{1}{m}
  Eg. Let p - large prime, a ∈ [1..p-1]
  b ∈ [0..m-1] all random
  \[ h_{a,b}(x) = ((ax + b) \mod p) \mod m \]

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

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

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
Overview:
- Separate Chaining
- Open Addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Linear probing: fast/slow
  - Quadratic probing: fast/slow
  - Hashing: complex

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: \( x \rightarrow \text{table}[h(x)] = \text{null} \)
- Alloc. new table size = \( n/\lambda_0 \)
- Compute new hash fn \( h \)
- Copy each \( x, v \) from old to new using \( h \)
- Delete old table

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>d</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>p</td>
<td>3</td>
</tr>
<tr>
<td>w</td>
<td>4</td>
</tr>
<tr>
<td>f</td>
<td>7</td>
</tr>
<tr>
<td>o</td>
<td>6</td>
</tr>
<tr>
<td>o</td>
<td>0</td>
</tr>
</tbody>
</table>

Table:
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

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

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

How to control \( \lambda \)?
- Rehashing: If table is too dense/too sparse, reallocate to new table of ideal size

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

Proof:
- On avg. each list has \( n/m = \lambda \)
  - Success: 1 for head + half the list
  - Unsuccess: 1 " " + all the list

If \( \lambda < \lambda_{\min} \) or \( \lambda > \lambda_{\max} \) ...
Open Addressing:
- Special entry ("empty") means this slot is unoccupied.
- Assume $\lambda \leq 1$
- To insert key:
  - Check: $h(x)$ if not empty try
    $h(x)+i_1$
    $h(x)+i_2$
  - Probe sequence: $\{i, i_1, i_2, \ldots\}$
- What's the best probe sequence?

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

Open Addressing

Hashing III

Analysis: Improves secondary clustering
- May fail to find empty entry
  (Try $m = 4$, $j^2 \mod 4 = 0 \cdot 1$ but not $2 \cdot 2$)
- 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.

Linear Probing:
- $h(x), h(x)+1, h(x)+2, \ldots$
  - Simple, but is it good?

Simple, but is it good?

$X: d, z, p, w, t$

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

Table $\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}$

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

Let $S_{ULP} = \frac{1}{2} (1 + \frac{1}{1 - \lambda})$

$U_{ULP} = \frac{1}{2} (1 + \frac{1}{1 - \lambda})^2$

Thm: $S_{ULP}$

Obs: As $\lambda \to 1$, time increase rapidly.

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

Quadratic Probing:
- $h(x), h(x)+1, h(x)+2, \ldots$
  - $t$ did not collide directly but had to probe 3 times!

Thm: $S_{ULP}$

$S_{ULP} = \frac{1}{2} (1 + \frac{1}{1 - \lambda})$

$U_{ULP} = \frac{1}{2} (1 + \frac{1}{1 - \lambda})^2$

Obs: As $\lambda \to 1$, time increase rapidly.
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]

Recap:

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

Open Addressing:

- Linear probing: \{ clustering
- Quadratic probing:

Hashing IV

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

- Proof is nontrivial (skip)

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>0.5</th>
<th>0.75</th>
<th>0.95</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_{DH} )</td>
<td>1.39</td>
<td>1.89</td>
<td>3.15</td>
<td>4.65</td>
</tr>
<tr>
<td>( U_{DH} )</td>
<td>2.00</td>
<td>4.00</td>
<td>20.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

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!)

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

Problem:
insert \( (a) \):
find \( (a) \):
delete \( (a) \):

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

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:

- \( E_{DH} \) = Expected search time of double hash. if successful
- \( U_{DH} \) = Exp. if unsuccessful

Recall: Load factor \( \lambda = n/m \)
Range Tree Applications:
- Range trees can be applied to a variety of query problems

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

Minimization/Maximization -

3-Sided Min Query
Given a set \( P \) of \( n \) pts in \( IR^2 \), 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 \( IR^2 \), 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 \( IR^2 \) and scalar \( l > 0 \), a NE triangle is a 45-45-90 right triangle with lower left corner q and side length \( l \).

Return a count of the number of pts of \( P \) lying within the triangle.
3-Sided Min Query

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

Data structure:
- Build a range tree for \( x \)
- Aux. trees are range trees for \( y \) that support find-larger

Query processing:
- Do 2D 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:

Return lowest in region \((x^+, y^+)\)

Data structure:
- Build a range tree for \( x \)
- Aux. trees are range trees for \( y \) that support find-larger

Query processing:
- Do ID 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: \( O(\log^2 n) \)

Transform coordinates to make orthog range query

\[ q^- = (x^-, y^-) \]

\[ q^+ = (x^+, y^+) \]

Line equation:
\[ y = x + (q_y - q_x^-) \]

Map each \( p = (p_x, p_y) \in \mathbb{P} \) to \( p' = (p_x', p_y') \) where \( p_x' = p_x \)

Let \( \mathbb{P}' \) be resulting set.

Build std. range tree for \( \mathbb{P}' \). Return ans. to query.

- \( q^- \leq x \leq q^+ \)
- \( q^-_y \leq y \leq q^+_y \)
- \( q^-_y - q^-_x \leq y \leq q^+_y - q^+_x \)
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? Can we do better?

Range Trees:
- Space is $O(n \log n)$
- Query time:
  - Counting: $O(\log n)$
  - Reporting: $O(k + \log n)$

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

→ Range trees are more limited

Recap:
- kd-Tree: General-purpose data structure for pts in $\mathbb{R}^d$
- Orthogonal range query:
  - Count/report pts in axis-aligned rect.
  - kd-Tree: Counting: $O(\sqrt{n})$ time
  - Reporting: $O(\sqrt{n} \log n)$ time

Call this a 1-Dim Range Tree:
Claim: A 1-Dim range tree with $n$ pts has space $O(n)$ and answers 1-D range count/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 w. Age range: $a_o \leq age \leq a_i$
  - Design a data structure for each criterion individually
  - Layer these structures together to answer full query

→ Multi-Layer Data-Structures

Range Trees I

Canonical Subsets:
- Goal: Express answer as disjoint union of subsets
- Method: Search for $Q_{i_0} + Q_{i_1}$

1-Dim Range Tree:
- Goal: Express answer as disjoint union of subsets
- Method: Search for $Q_{i_0} + Q_{i_1}$
  - Balanced BST (e.g. AVL, RB, ...)
  - Assume extended tree
  - Each node $p$ stores no. of entries in subtree: $p.size$

Design a data structure for the criteria individually
- Canonical Subsets:
Recursive helper:
```c
int range1Dx(Node p, Intv Q, Intv C=[x0, x1])
```

Initial call: `range1Dx(root, Q, C)`

Cases:
1. **p is external:**
   - if `p.pt.x ∈ Q` → 1 else → 0
2. **p is internal:**
   - `C ⊆ Q` ⇒ all of p's pts lie within query
     → return p.size
   - `C is disjoint from Q` ⇒ none of p's pts lie in Q
     → return 0
   - Else partial overlap
     → Recurse on p's children + trim the cell

More details:
Given a 1-D range tree T:
- Let `Q=[Q0, Q1]` be query interval

For each node p, define
- `interval cell C=[x0, x1]`
- s.t. all pts of p's subtree lie in C

Root cell: `C0=[-∞, +∞]`

Range Trees II

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

Thm: Given 1-D range tree... can answer range queries in time O(log n) → (k to report)
Answering Queries?

Given query range
\[Q = [Q_{lo.x}, Q_{hi.x}] \times [Q_{lo.y}, Q_{hi.y}]\]
- Run 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:
  \[\log n \times \log n \times \cdots \times \log n = O(\log^d n)\]

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

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

**Digital Search**

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

**Example:** $\Sigma = \{a = 0, b = 1, c = 2\}$
- Keys: $\{aab, aba, abc, caa, cab, cbc\}$

**Analysis:**

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

**Example:**

![Diagram of Tries and Digital Search Trees]

**How to save space?**

- de la Briandais trees:
  - Store 1 char. per node
  - $x \rightarrow \# x \Rightarrow$ try next char in $\Sigma'$
  - $= x \Rightarrow$ advance to next character of search string
- First-child/next-sibling

**Node:** Multiway of order $k$

![Diagram of Node Structure]

**Example:**

- Same structure/Alt. Drawing

**Search:** $\sim$ length of query string [O(1) time per node]

**Space:**
- No. of nodes $\sim$ total no. of chars in all strings
- Space $\sim k \cdot$ (no. of nodes)
Patricia Tries:
- Improves trie by compressing degenerate paths
- PATRICIA = Practical Alg. to Retrieve Info. Coded in Alpha
- Late 1960's: Morrison & Guchenberger
- Each node has index field, indicates which char to check next (Increase with depth)

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

Example: $S_0: \text{ajam}, \text{aj}$
$S_1: \text{pajam}, \text{paj}$
$S_2: \text{mapaj}, \text{map}$
$S_3: \text{apaj}, \text{ap}$
$S_4: \text{ama}, \text{ama}$
$S_5: \text{jama}, \text{j}$
$S_6: \text{pamapa}, \text{pam}$

Tries and Digital Search Trees II

Example: $S = 0123456789$
- $S_0: \text{pamapa}, \text{pam}$
- $S_1: \text{apaja}, \text{ap}$

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

Analysis:
- Query time: (Same as std trie) \* search string length (may be less)
- Space:
  - No. nodes \* No. of strings (irres. of length)
  - Total space: $K \cdot (\text{No. of nodes}) + \text{(Storage for strings)}$

Def: Substring identifier for $S_i$: is shortest prefix of $S_i$ unique to this string
$S_i$ = $a_{i-1}, a_{i-2}, \ldots, a_0$
$S_i$ = substring needed to identify suffix $S_i$?
Example: \( S = \text{pumapajama} \)

Suffix Trees (cont.)

- \( S \) - text string \( |S| = n \)
- \( S_i \) = \( i^{th} \) suffix

Substring ID = min substr. needed to identify \( S_i \)

A suffix tree is a Patricia trie of the \( n+1 \) substring identifiers

Example: \( \text{ID}(S_i) = \text{amap} \) \( \text{ID}(S_j) = \text{ama} \)

Substring Queries:

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

Tries and Digital Search Trees III

- Search time: \( n \) total length of target string
- Construction time: \( O(n \cdot k) \) [nontrivial]

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

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

Assume points lie in unit square

Geometric Applications:

Analysis:

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

Example:

Search("ama") \( \rightarrow \) End at intern node \( \text{ama} \) \( \rightarrow \) End at intern node \( \text{ama} \)

Search("amapaj") \( \rightarrow \) End at extern node \( \text{ama} \)

Goto \( S_i \), verify

Report: 2 occ.

Final tree:
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 a binary fraction:
  \(x = 0.a_1a_2a_3\ldots, a_i \in \{0, 1\}\)
  \(x = \sum a_i \cdot 2^{-i}\)

Example:

<table>
<thead>
<tr>
<th>(x)</th>
<th>(0.0)</th>
<th>(0.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.0)</td>
<td>(0.1)</td>
<td></td>
</tr>
<tr>
<td>(0.01)</td>
<td>(0.10)</td>
<td></td>
</tr>
<tr>
<td>(0.010)</td>
<td>(0.101)</td>
<td></td>
</tr>
</tbody>
</table>

\(\text{PR kd-Tree} \equiv \text{Trie}??\)

- 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 \(\text{kd-tree}\)

Further Remarks:
- Techniques for efficiently encoding, building, serializing, compressing...
  tries apply immediately to \(\text{PR kd-tree}\)
- Can generalize to any dimension

\(x = 0.a_1a_2a_3\ldots\)
\(y = 0.b_1b_2b_3\ldots\)
\(z = 0.c_1c_2c_3\ldots\)

Lemma: Given a point set \(P \subseteq \mathbb{R}^n\)
  (in unit square \([0, 1]^n\)) 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 \(\text{PR kd-tree}\) for \(P\) is equivalent to binary trie for \(\Phi(P)\).

Example:

How do we extend to 2-D?

<table>
<thead>
<tr>
<th>(y = 0)</th>
<th>(y = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>(1)</td>
</tr>
<tr>
<td>(0)</td>
<td>(1)</td>
</tr>
<tr>
<td>(0)</td>
<td>(1)</td>
</tr>
<tr>
<td>(0)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

**PR kd-tree**

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

**Morton Code of** \(p\)

Define:
\[
\Phi(x, y) = a_1b_1a_2b_2a_3b_3\ldots
\]

Called **Morton Code** of \(p\)

Proof: By induction on no. of bits

<table>
<thead>
<tr>
<th>(x)</th>
<th>(0)</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>(0)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

The \(\text{PR kd-tree}\)

+ Binary trie assigns pts to same subtrees
  \(\ldots\) induction
Deallocation Models:
Explicit: (C++/C++)
- programmer deletes
- may result in leaks, if not careful
Implicit: (Java, Python)
- runtime system deletes
  - Garbage collection
  - Slower runtime
  - Better memory compaction

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 / heaport

Block Structure:
Allocated:
- inUse
- prevInUse
Available:
- inUse
- prevInUse

Memory Management

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

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

Guide:
- prevInUse: 1 if prev. contiguous block is allocated
- prev/next: links in avail. list
- size, size2: total block size (includes headers)

How to select from available blocks?
- First-fit: Take first block from avail. list that is large enough
- Best fit: Find closest fit from avail. list
- Surprise: First-fit is usually better - faster & avoids small fragments

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

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Allocation:** `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: merge with either/both to make max. avail 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.inUse`, `p.prevInUse`, `p.size`
  - We omit bit manipulation
  - \( *(p+p.size-1) \) - references last word in this block

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

```c
void* p = NULL; // initialize pointer
```
Buddy System:
- Block sizes (including headers) are power of 2
- Requests are rounded up (internal fragmentation)
- Block size $2^k$ starts at address that is multiple of $2^k$
- $k = \text{level of a block}$

Structure:

<table>
<thead>
<tr>
<th>Level</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>$0, 3, 15$</td>
</tr>
<tr>
<td>3</td>
<td>$0, 3, 7, 10$</td>
</tr>
<tr>
<td>2</td>
<td>$0, 3, 4, 6, 8, 11$</td>
</tr>
<tr>
<td>1</td>
<td>$0, 3, 4, 5, 7, 8, 10, 11$</td>
</tr>
<tr>
<td>0</td>
<td>$0, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$</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 frag at cost of internal frag.

Memory Management

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

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

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

Big Picture:
- Avail list is organized by level: avail[$k$]
- Block header structure same as before except:
  prevInUse $\neq$ not needed
  size = 2

Example: $\text{alloc}(2) \rightarrow \text{alloc}(4)$
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 decrease-key?
- 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 increase-key?
  - Heaps are very asymmetrical w.r.t. keys.

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

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

Quake Heaps I:
- Basic definitions
- Operations

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

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 increase-key?
  - Heaps are very asymmetrical w.r.t. keys.

History:
- 1984: Fibonacci Heaps (Fredman + Tarjan)
  - Many variants
  - Complex to analyze
- 2013: Quake Heap (Timothy Chan)
  - Much simpler
**Basic utilities:**

- `make-root(Node u)`: Make `u` a root
- `trivial-tree(Key x)`: Create 1-node tree with `key x`
- `link(Node u, Node v)`: Link `u` + `v` roots on same level

**Quake Heaps II**

- **Utility ops**
  - `Insert`
  - `Decrease-key`

**void cut(Node w)**

Node `w` ← `w`.right
if (`v` ≠ null)
  if (`v`.key ≤ `w`.key)
    `w`.right ← null
  `make-root(v)`
else
  `w`.right ← `v`
`w`.parent ← `v`.parent ← `w`
return `w`

**void decrease-key(Locator r, Key y)**

Node `u` ← `r`.get Node() // get leaf node
Node `u`.child ← null
do {
  `u`.key ← `y` // update key value
  `u`.child ← `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
- `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`)
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 @QUAKE!

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

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

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

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

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

Quake Heaps
- Change key

Void delete-left-path (u)
- while (u \neq null)
  - cut (u)
  - nodeCt [u.level] = 1
  - u = u.left

Void merge-trees ()
- for (lev = 0 .. nLevels - 2)
  - while (roots [lev].size \geq 2)
    - Node u, v \leftarrow remove any 2
    - make-root (link (u, v))
    - if (nodeCt [lev+1] > 3/4 \cdot nodeCt [lev])
      - clear-all-above (lev)

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

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

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)

Times:
- Insert - \(O(1)\)
- Decrease-key - \(O(\log n)\)
- Extract-min - ??

Can we do better? \(O(1)\)?

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.

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.

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].

Quake Heaps V
- Analysis (Quick + Dirty)

Quake Heap Potential:
- Let $N = \text{no. of nodes}$
- $R = \text{no. of roots}$
- $B = \text{no. of nodes with}$
- \hspace{2mm} 1 child (bad nodes).

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

Idea: The amortized cost of an operation defined to be $(\text{actual-cost}) + (\text{change in } \Psi)$.

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

Lemma: Amortized cost of
- insert/dec-key = $O(1)$
- extract-min = $O(\log n)$
Minimum Spanning Trees:
- Given a connected, weighted graph \( G = (V, E) \)
  \[ (u, v) \in E \rightarrow w(u, v) = \text{weight} \]

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

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

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

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 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 \( (p_i, p_j) \) \( p_i \in S \), \( p_j \in P \setminus S \)

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

Euclidean MST (EMST)
- The MST of \( P \)’s

Euclidean Graph:
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:
\[ w(p_i, p_j) = \text{dist}(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \]
Finding next edge?
- Brute force: $O(n^2)$
- \( P \) (Points B. Point start)

Prim’s Algorithm:
- Given point set \( P \) + start pt \( s_0 \)
  - \( S \in P \): Pts in spanning tree
    - Init: \( S = \{s_0\} \) End: \( S = P \)
  - \( P \setminus S \): Pts not yet in tree
    - While \( (S \neq P) \)
      - Find closest \((p_i, p_j) \in P \setminus S \)
      - Add \( p_i \) to \( S \)
      - Add \((p_i, p_j)\) to tree

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

\( S = \) \{SFO, DFW, ORO, JFK, LAX\}
\( P \setminus S = \) \{IAD, ATL, SEA\}

Euclidean MSTs (II)

How to do this?
- Lots of data structures!

Basic Objects:
- \( \text{edgeList} \): list of edges in tree
- \( \text{inEMST} \): set representing \( S \)
- \( \text{kdTree, heap} \):
- \( \text{dependents} \): dep lists for all \( P \setminus S \)

Priority Queue: Stores the NN pairs ordered by squared dist.
(Eg. \((SFO,BWI)\), \((DFW,ORD)\),...)

List: Store edges of tree
(Eg. \{(SFO,DFW), (DFW,ORD),...\})

Set: Store points of \( S \)
(Eg. \{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

Dependents list \( \text{dep}(p_j) \)
is list of all pts \( p_i \) that depend on \( p_j \)
add Edge (Pair (Point) edge) =>
add edge to edge List (first, second)
pt2 ← edge.getSecond()
add pt2 to in EMST
delete pt2 from kdTree
dep2 ← get pt2 dep list from dependents
for each (pt3 in dep2)
    mn3 ← kdTree.near Neigh (pt3)
    if (mn3 == null) break
    add NN (pt3, mn3)

Helpers:
- Initialize (Point start)
  - initialize all structures
  - add edge (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 u. 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

Is this efficient?
- Assuming NN queries in O (log n) time
  Total time = O (n log n + m log n)
m = # of NN updates
  Much depends on m
  m depends on pt. distrib.