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

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

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

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

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

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 \(a_1, a_2, ..., a_n\). Operations:
- `init()` - create an empty list
- `get(i)` - returns \(a_i\)
- `set(i, x)` - sets \(i^{th}\) element to \(x\)
- `insert(i, x)` - inserts \(x\) prior to \(i^{th}\) (moving others back)
- `delete(i)` - deletes \(i^{th}\) item (moving others up)
- `length()` - returns num. of items

Implementations:
- **Sequential**: Store items in an array
  
  \[
  \begin{array}{c}
  a_1, a_2, a_3, ..., a_n \\
  \end{array}
  \]
- **Linked**: Store items in a linked list
  
  \[
  \text{Singly: } \quad \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{null}
  \]
  
  \[
  \text{Doubly: } \quad \text{head} \rightarrow a_1 \leftarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{null}
  \]

  Performance varies with implementation

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

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

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

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

Stack: All access from one side
- Push + pop

Queue: First In First Out (FIFO) list: enqueue inserts at tail and dequeue deletes from head
Cost model (Actual cost) ↔ Dynamic (Sequential) Allocation

Cheap: No reallocation → 1 unit
Expensive: Array of size n → 2n+1

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 ≤ 5m]

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 → run
  1 2 3 4 5 6 7 8 9 10 11 12
  - At start of a run there are n+1 items in stack and array size is 2n
  - There are at least n ops before the end of run
- During this time we collect at least 5n tokens → 1 for each op → 4 for deposit
- Next reallocation costs 4n, but we have enough saved!

Actual tokens: +5 +11 +5 +11 +17 = 28
Tokens: +5 +5 +5 +5 = 20
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)
\[ n \rightarrow n^2 \text{ or } n^{1.57} \]

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

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

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

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

Multilists: Lists of lists

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

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

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

Depth: path length from root

Height: (of tree) max depth

Degree (of node): number of children

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

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

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

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

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

Node structure:
- Each node stores a data and pointers to its children

Trees Representation + Binary Trees (I)

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

Full: Every non-leaf node has 2 children

Full: Every non-leaf node has 2 children

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

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

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

Binary Tree Traversals (can be generalized)

root ... \( \rightarrow \)
- 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

Another way to save space...

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

Eg. Inorder Threads:
Null left \( \rightarrow \) inorder predecessor
Null right \( \rightarrow ^* \) successor

Those wasteful null links....

In order:
\[ a \ b \ c \ + \ d \ e \]
Examples:
- Given prime \( p \), \( a \equiv b \) mod \( p \)
  \[ \text{Eg: } p=5 ; \text{Partition: } \{0,5,10,...,\};\{1,6,11,...\} \]
- Given graph \( G \), vertices \( u, v \),
  \( u \equiv v \) if in same connected component

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

Any equivalence relation defines a partition over \( S \).

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

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

Set Identifiers are indices of root nodes

Eg.
\[
\begin{align*}
\text{Find(7)} &= 3 \\
\text{Find(10)} &= 3 \\
\text{Find(5)} &= 2
\end{align*}
\]

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

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

parent \([1..n]\), where \( \text{parent}[i] \) is \( i \)'s parent index or \( \text{O} = \text{null} \) if root

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

Eg.
\[
\begin{align*}
\{1,3,7,10\} \Rightarrow \{2,5,6,8,11\} \Rightarrow \{9,9\}
\end{align*}
\]

Disjoint Set Union-Find I
Set Union \((\text{Set } s, \text{Set } t)\) \{ 
\text{if } (\text{rank}[s] > \text{rank}[t]) \}
[swap s \leftrightarrow t]
parent[s] \leftarrow t
rank[t] \leftarrow \max (\text{rank}[t], 1 + \text{rank}[s])
\}
return t

Set Union

\[\begin{align*}
\text{Recall: These are just array indices of roots} \\
\text{Just link one tree under the other} \\
\text{How to maintain low} \\
\text{Rank: Based on height of tree. Link lower} \\
\text{rank as child}
\end{align*}\]

Init: All ranks \(\leftarrow 0\)

Example:
\[\{1, 3, 7, 10\}, \{1, 2, 5, 6, 8, 11\}, \{4, 9\}\]

Union \((9, 12)\) 
[\(\text{r}2\) has lower rank]
\[\text{rank}[9] = \max (\text{rank}[9], 1 + \text{rank}[12]) = 1\]

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

Running Time?
Init: \(O(n)\) - set a parents to null + ranks to \(0\)
Union: \(O(1)\) - constant time
Find: \(O(\text{tree height})\)

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

Disjoint Set Union - Find II

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

Proof: By induction on num. of unions
Basis: Single node. \(h = 0\), \(2^0 = 1\) nodes
Step: Consider the last of series of unions. Let \(T' + T''\) be trees to merge: Heights: \(h' + h''\)
Sizes: \(n' + n''\)
By induction: \(n' \geq 2^{h'}\), \(n'' \geq 2^{h''}\)
Cases: \(h' = h''\)
Cases 2: \(h' < h''\)
Cases 3: \(h' > h''\) (symmetrical)

Running Time?
Init: \(O(n)\) - set a parents to null + ranks to \(0\)
Union: \(O(1)\) - constant time
Find: \(O(\text{tree height})\)

\[\text{What is worst case?}\]
We'll show tree height = \(O(\log n)\) 
\[\Rightarrow \text{Find takes } O(\log n) \text{ time}\]
Path Compression:
- Whenever we perform find, short-cut the links so they point directly to root.
- This does not increase running by more than constant, but can speed up later finds.

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

$$\Rightarrow \text{Amortized time (average per op) is } O(\log m)$$

- Not bad - But can we do better?

Example:

```
      5
     / \
    4   3
   /   / \ \
  2   4   3
 /    /   / \
1    2   1
```

Disjoint Set Union-Find III

Digression: Ackerman's Function (1926)

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

Does this little trick improve running times?

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

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

$$\Rightarrow \text{Amortized time } = O(\alpha(m, n))$$

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

From super big to super small
Inverse of Ackerman

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

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

Digression: Ackerman's Function

$$A(0, j) = \begin{cases} 
  1 & \text{if } j = 0 \\
  2j+2 & \text{o.w.}
\end{cases}$$

Looks innocent, but it's a monster!

More than atoms in universe
**Naive Solution:**
- Store items in linear list
- Order?

**Insert order** - fast insert/slow extract
**Priority order** - fast extract/slow insert

**Heap**: Tree-based structure

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

[Reverse: max-heap order]

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

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

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

**Priority Queues + Heaps I**

**Heap**:
- Tree-based structure

**Heap Structure**
- Root has no parent
- Left child ≤ right child
- Parent’s key ≤ both children's keys

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

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

**void insert(key x)**
- $n++$; $i \leftarrow$ sift-up($n,x$)
- $A[i] \leftarrow x$

**int sift-up(int i, key x)**
- while ($i > 1$ && $x < A[par(i)]$)
- $A[i] \leftarrow A[par(i)]$
- $i \leftarrow par(i)$
- return $i$

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

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

Leftist Property: Null path length
\[ npl(v) = \begin{cases} -1 & \text{if } v = \text{null} \\ 1 + \min \left( npl(v.\text{left}) \right) & \text{otherwise} \end{cases} \]

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

Examples

leftist heap: meldable heaps
- can merge two heaps into single heap
- eg. One processor breaks.
  - awaiting jobs must be merged with another processor.

Analysis: Both insert + extract-min take time proportional to tree height
- tree is complete \( \Rightarrow O(\log n) \) time
Class structure:

Letist Heap $\langle \text{Key} \rangle$

private class LHNode {
    Key x
    LHNode left, right
    int npl
}

private LHNode root
public LeftistHeap () {root = null}
    "constructor"
    public functions
    ... (other private/protected utilities)

public mergeWith (LeftistHeap H2) {
    root ← merge (this.root, H2.root)
    H2.root ← null
    "helper function" merge destroys H2
}

Merge helper: 2 phases

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

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

Proof: (Sketch - see latex notes)

Analysis: Time $= O(\log n)$

- Insert + Extract-min
- Exercises

LHNode merge(LHNode u, LHNode v)
if (u == null) return v
if (v == null) return u
if (u.key > v.key) // swap so u is smaller
    swap u ← v
if (u.left == null) u.left ← v
else
    u.right ← merge(u.right, v)
    if (u.left.npl < u.right.npl)
        swap u.left ← u.right
        u.npl ← u.right.npl + 1
    return u
Dictionary:
- **insert** (Key x, Value v)
  - Insert (x, v) in dict. (No duplicates)
- **delete** (Key x)
  - Delete x from dict. (Error if x not there)
- **find** (Key x)
  - Returns a reference to associated value v, or null if not there.

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

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

**Binary Search Trees**

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

**Example:**
- root ...
- find(14)

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

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

**Dictionary:**
- 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: <, >, =

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

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

**Binary Search Trees**

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

Why did we do:
- p.left = insert (x, v, p.left)

3 cases:
1. x is a leaf
2. x has single child
3. x has two children

Find replacement node → copy to x, and then delete x
BSTNode delete (Key x, BSTNode p)
    if (p == null) error! Key not found
    else if (x < p.key)
        p.left = delete (x, p.left)
    else if (x > p.key)
        p.right = delete (x, p.right)
    else if (either p.left or p.right null)
        if (p.left == null)
            return p.right
        if (p.right == null)
            return p.left
    else
        r = find Replacement (p)
        copy r's contents to p
        p.right = delete (r.key, p.right)
    return p

BSTNode find Replacement (BSTNode p)
    BSTNode r = p.right
    while (r.left != null)
        r = r.left
    return r

**Binary Search Trees III**
- deletion
- analysis
- Java

**Java Implementation:**
- Parameterize Key + Value types: extend Comparable
  - class BinSearchTree<K,V>
- BSTNode - inner class
- Private data: BSTNode root
- insert, delete, find: local
- provide public ens
  - insert, delete, find

But height can vary from $O(\log n)$ to $O(n)$...

Expected case is good

**Thm:** If $n$ keys are inserted in random order, expected height is $O(\log n)$.

**Analysis:**
All operations (find, insert, delete) run in $O(h)$ time, where $h$ = tree's height
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...
    }

    private Node root;

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

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

    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 \( \pm 1 \).

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 \( h \))
- \( h = 0: n(h) = 1 = F_2 - 1 \)
- \( h = 1: n(h) = 2 = F_3 - 1 \)
- \( h \geq 2: n(h) = n(h-1) + n(h-2) \)
  \[ n(h) = 1 + n(h-1) + n(h-2) = 1 + (F_{h-2} + F_{h-1}) = (F_{h+2} + F_{h+1}) - 1 = F_{h+3} - 1 \]

Corollary: An AVL tree with \( n \) nodes has height \( O(\log \sqrt{n}) \).
Proof: Fact: \( F_h \approx \phi^h / \sqrt{5} \)
- \( n \geq \phi^h = c \cdot \phi^h \Rightarrow h \leq \log \phi + c' \)
  \[ h \leq \log \frac{n}{\log \phi} + c' \]
  \[ \Rightarrow n \geq \phi^{\log \phi \cdot n} \log \phi \]
  \[ = O(\log n) \]
Double rotations:
- Left-right
- Right-left

AVL Trees II
- double rotations
- insertion

How to rebalance?
Bal = -2
Left-right heavy

Left-right heavy:

Utilities:
- int height(AVLNode p)
  return \( \begin{cases} 
  p == \text{null} & \rightarrow -1 \\
  \text{otherwise} & \rightarrow p.\text{height} 
\end{cases} \)
- void upadateheight(AVLNode p)
  p.\text{height} = 1 + \text{max}(\text{height}(p.\text{left}), \text{height}(p.\text{right}))
- int balanceFactor(AVLNode p)
  return \text{height}(p.\text{right}) - \text{height}(p.\text{left})

AVLNode rebalance(AVLNode p)
if (p == null) return p
if (balanceFactor(p) < -1)
  if (ht(p.\text{left}.\text{left}) >= \text{ht}(p.\text{left}.\text{right})
      p = rotateRight(p)
    else p = rotateLeftRight(p)
  else if (\text{balance Factor}(p) > +1)
    \text{...symmetrical}
  updateHeight(p); return p

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

- Balance factor -2
  - Left-left heavy
    - Apply standard BST deletion
    - find key to delete
    - find replacement node
    - copy contents
    - delete replacement
    - rebalance

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

**Deletion:** Basic plan

1. Right rotation
2. find key to delete
3. find replacement node
4. copy contents
5. delete replacement
6. rebalance

**AVL Trees III**

- Deletion
- Examples

**AVL-Node delete (Key x, AVLNode p):**

- same as BST delete
- return rebalance(p)

**Examples:**

- Insert(8)
- Insert(5)
- LR rotate
- Rotate right
- LR rotate
- Rotate right
Node types:
- 2-Node
  - 1 key
  - 2 children
- 3-Node
  - 2 keys
  - 3 children

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

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

2-3 Trees

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 to 4-nodes temporary

Adoption
(Key Rotation)

1+3 = 2+2

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

Split:
$4 \rightarrow 2+2$

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

Example:
2-3 tree of height 2

2-3 Trees

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)
Dictionary operations:

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

Implementation?

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

2-3 Trees II

Delete Example:

```
delete(5)
```

Insertion example:

```
insert(6)
```

Deletion remedy:

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

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

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

Example:
2-3 Tree:

Red-Black:

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

What we need are stricter rules!

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) Yes!
B) No!

A "left-skewed" encoding
Corresponds to 2-3-4 trees
Restructuring Ops:

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

  \[ \xrightarrow{b} \]

  How to test? \( p \cdot \text{left}\cdot\text{level} = p\cdot\text{level} \)

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

  \[ \xrightarrow{b} \]

  How to test? \( p\cdot\text{level} = p\cdot\text{right}\cdot\text{level} = p\cdot\text{right}\cdot\text{right}\cdot\text{level} \)

  not needed (levels are monotone)

**Example**

**2-3 Tree**: \( 4:11 \)

2-3 Tree → AA tree

- **AA tree**: \( 2 \)

  \[ \xrightarrow{11} \]

  **Level 3**

  \( 1 \)

  \( 15:20 \)

  \( 14:19 \)

  \( 13:51 \)

  \( 12:41 \)

  \( 1 \)

  \( 11:11 \)

  \( 10:21 \)

  \( 9:11 \)

  \( 8:11 \)

  \( 7:11 \)

  \( 6:11 \)

  \( 5:11 \)

  \( 4:11 \)

  all to nil

  \[ \xrightarrow{11} \]

**Red-Black + AA Trees II**

**AA Insertion**:

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

**AA Node skew(AANode p)**

\[
\begin{align*}
\text{if } (p = \text{nil}) & \text{ return } p \\
\text{if } (p\cdot\text{left}\cdot\text{level} = p\cdot\text{level}) & \text{ right rotate } p \\
\quad \text{AANode } q = p\cdot\text{left} \\
\quad p\cdot\text{left} = q\cdot\text{right}; q\cdot\text{right} = p \\
\quad \text{return } q \leftarrow \text{new subtree root} \\
\text{else } & \text{ return } p \leftarrow \text{everything is fine} \\
\end{align*}
\]

**AA Node split(AANode p)**

\[
\begin{align*}
\text{if } (p = \text{nil}) & \text{ return } p \\
\text{if } (p\cdot\text{right}\cdot\text{right}\cdot\text{level} = p\cdot\text{level}) & \text{ left rotate at } p \\
\quad \text{AANode } q = p\cdot\text{right} \\
\quad p\cdot\text{right} = q\cdot\text{left} \\
\quad q\cdot\text{left} = p \\
\quad q\cdot\text{level} += 1 \leftarrow \text{move } q \text{ up a level} \\
\quad \text{return } q \\
\text{else } & \text{ return } p \leftarrow \text{all okay} \\
\end{align*}
\]
**Red-Black and AA Trees III**

### Deletion

Two more helpers:

**UpdateLevel:** If p's level exceeds $l = 1 + \min(p.\text{left.level}, p.\text{right.level})$ then set p's level to $l + 1$ 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)
Randomized Data Structures
- Use a random number generator
- Running in expectation over all random choices
- Often simpler than deterministic

Geometric Interpretation:
- Key \( \rightarrow x \)
- Priority \( \rightarrow y \)

Example:
- Key
  - b: 37
  - c: 45
  - e: 3
  - f: 14
  - g: 51
  - h: 57
  - k: 45
  - m: 78
  - o: 67

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

? Is it always possible to do both?
Yes: Just consider the corresponding BST.

Intuition:
- Random insertion into BSTs \( \Rightarrow O(\log n) \) expected height
- Worst case can be very bad \( O(n) \) height

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

Along any path - Insertion times increase
**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 to put lowest priority node at $p$.

**Example:**

**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
  - Every other
  - Every fourth
  - Every $2^i$
  - Sorted linked lists:
    - Easy to code
    - Easy to insert/delete
    - Slow to search: $O(n)$
  - Idea: Add extra links to skip
  - How to generalize?

Example:

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

Node Structure:
- (Variable sized)
- class SkipNode{
  - Key key
  - Value value
  - SkipNode[] next
  - In constructor, set size (height)
}

Value find(Key x) {
  i = topmost level
  SkipNode p = head
  while (i ≥ 0) {
    if (p.next[i].key ≤ x) p = p.next[i]
    else i--; 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 $\frac{1}{n}$. Let $l = c \cdot \log n$.

1. **Prob that any given node's level exceeds $l$ is $\frac{1}{2^l}$** [l consecutive heads]
2. **Prob that any of $n$ node's level exceeds $l$ is $\frac{n}{2^l}$** [n trials with prob $\frac{1}{2^l}$]
3. **Let $l = c \cdot \log n$**
4. **Prob that max level exceeds $c \cdot \log n$ is**: $\frac{n}{n^c} = \frac{1}{n^{c-1}}$

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

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

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

- Let $n_i$ = no. of nodes that contrib. to level $i$.
- **Expected no. of nodes that contrib. to level $i = n/2^i$**
- **Prob that node at level $\geq i$ is $\frac{1}{2^i}$**

Let $n = \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$

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

**Cases:**

- **Case (A):** Current node $\uparrow$
- **Case (B):** Same level $\uparrow$
- **Case (C):** From prior level $\uparrow$

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

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

**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?)

- **Cases:**
  - **Case (A):** Current node $\uparrow$
  - **Case (B):** Same level $\uparrow$
  - **Case (C):** From prior level $\uparrow$

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

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

$E(l) = 2 + E(l-1)$

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

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

$\Rightarrow \text{We visit 2 nodes per level on average}$
### Delete:
- Start at top
- Search each level saving last node < key
- On reaching node at level 0, remove it and unlink from saved pointers

### Insert:
(Similar to linked lists)
- Start at top level
- At each level:
  - Advance to last node ≤ key
  - Save node + drop level
- At level 0:
  - Create new node (flip coin to determine height)
  - Link into each saved node

### Example: find(25)

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

#### Delete(12)
- visit, don’t save
- visit, save reference

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

### Note:
Variation in running times due to randomness only—not sequence.

\(\Rightarrow\) User cannot force poor performance.
**Other/Better Criteria?**
- Expected case: Some keys more popular than others
- Self-adjusting: Tree adapts as popularity changes

**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

**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

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

**Tree's height has reduced by ~ half!**

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

**Idea II:** Rotate 2 at a time - upper + lower
(still unbalanced)
ZigZig(p): [LL case]

\[ \text{Node } p \leftarrow \text{find } x \text{ by standard BST search} \]
\[ \text{while } (p \neq \text{root}) \{
\text{if } (p = \text{child of root}) \text{ zig}(p)
\text{else /* p has grand parent */}
\text{if } (p \text{ is LL or RR grand child}) \text{ zigZig}(p)
\text{else /* p is LR or RL gr. child */ zigZag}(p)
\}
\]

Inserted node:
\[ \text{node } p \leftarrow \text{play(x)} \]
\[ \text{if } (p.\text{key} == x) \text{ Error!!} \]
\[ \text{q} \leftarrow \text{new Node}(x) \]
\[ \text{if } (p.\text{key} < x) \]
\[ p.\text{left} \leftarrow p \]
\[ p.\text{right} \leftarrow p.\text{right} \]
\[ \text{else ... (symmetrical) ...} \]
\[ \text{root} \leftarrow q \]

Example:
\[ \text{root} \leftarrow \text{play(x)} \]
\[ \text{if } (\text{root.key} == x) \]
\[ \text{return root.value} \]
\[ \text{else return null} \]

Splay Trees II

Subtrees A,C move up

Subtrees C,E of p move up

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 $\Delta_j = i_j - i_{j-1}$, 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 $H = \sum p_i \log \frac{1}{p_i}$ is the entropy.

**Static Optimal Tree 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 $\pi_i = q_i/m$. Then total time is $O(m \sum \pi_i \log \frac{1}{\pi_i})$.

**Splay Trees:**

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

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

- **Splay Trees are Amazingly Adaptive!**
Geometric Search:
- Nearest neighbors
- Range searching
- Point Location
  - Intersection Search

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

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

Quadtrees & kd-Trees

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

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

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

- Example:
  - (5, 4)
  - (2, 2)
  - (1, 3)
  - (4, 1)

**Quadtree:**
- Each internal node stores a point.
- Cell is split by horizon + vertical lines through point.

- Example:
  - (5, 4)
  - (2, 2)
  - (1, 3)
  - (4, 1)

**Quadtree Analysis:**
- 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).

**Quadtree History:**
- Bentley 1975
- Called it 2-d tree (\( \mathbb{R}^2 \))
- In short: \( \text{kd-tree} \) (any dim)
- Where/which direction to split? 
  - Next

**kd-Tree:**
- Binary variant of quadtree
- Splitter: Horizon or vertical line in 2-d (orthogonal plane ow.)
- Cell: Still AABB

- Subcells:
  - Left: left/below/above

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

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

Example:
```java
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
```java
find(Point q, KDNode p) {
    if (p == null) return null;
    if (q == p.point) return p.value;
    if (q [cutDim] < p.point [cutDim]) return find(q, p.left);
    else if (q.onLeft(p)) return find(q, p.left);
    else return find(q, p.right);
}
```

Quad-trees & Kd-Trees III

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

Helper:
```java
class KDNode {
    boolean onLeft(Point q) { return q[cutDim] < pt[cutDim] }
}
```
KD-Tree Insertion:

(Similar to std. BSTs)

- Descend tree until finding path to leaf
  - If found: leaf node → just remove
  - Internal node → find replacement
    - Copy here
    - Recur. delete replacement

- If not found:
  - Descend path to leaf
  - If cut along dimension:
    - Create new node
      - Set cutting dimension

Rebalance by Rebuilding:

- Rebuild subtrees
  - As with scapegoat trees
  - $O(h)$ amortized
  - Find: $O(h)$ guaranteed

Analysis:

- Run time: $O(h)$

Can we balance the tree?

- Rotation does not make sense!!
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 $g$

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

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

Kd-Tree Queries

Rectangle methods for kd-cells:
- Split a cell $r$ by a split pt $s \in r$, along cut dim cd
- $r$ high
  - left part
  - right part
- $r$.leftPart(cd, s)
  - returns rect with $lw = r$.lw + high = $r$.high but $high[cd] \leftarrow s[cd]$
- $r$.rightPart(cd, s)
  - $high = r$.high + low = $r$.low but $low[cd] \leftarrow s[cd]$

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

Useful methods:
- Let $r, c$ - Rectangle
- $q$ - Point
- $r$.contains($q$)
- $r$.contains($c$)
- $r$.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
  → 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**

```
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++ // pt in range
        ct += rangeCount(R, p.left, cell.leftPart(p.cutDim, p.pt))
        ct += rangeCount(R, p.right, cell.rightPart(...)
        return ct
    }
}
```

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

For us: $a = 2$
$$\log_b 4 = \frac{d}{d-1} = \frac{n^{1/d}}{n^{1/d-1}}$$

$$= n^{1/d} = \sqrt[3]{n}$$

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

Recurrence:
$$T(n) = 2 + 2T(n^{1/4})$$

Lemma: Given a Kd-tree (as in Thm above) and horiz. or vert. line $l$, at most $O(n^{1/d})$ cells can be stabbed by $l$.

Proof: w.l.o.g. $l \parallel$ horiz.

Cases: $p$ splits vertically
- $p$, $l$, stab both
- $p$, $l$, stab 1
- $p$ splits horizontally
- $p$, $l$, stab only one

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

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

Simpler: Extend $R$'s sides to 4 lines and analyze each one.
Range Tree Applications:
- Range trees can be applied to a variety of query problems

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

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

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 triange 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
regin $x_0 \leq x \leq x_1$, $y \geq y_0$

Skewed rectangle query:
Transform coordinates to make orthog range query

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: $O(\log^2 n)$

Line equation:
$q_x = (x^+, y^+)$
$q^- = (x^-, y^-)$

$\Rightarrow q^- - q_x \leq p_y - p_x \leq q^+ - q_x$
$p_y = p_y - p_x$
$p_x \leq p_x
Let $P'$ be resulting set.

$\Rightarrow q^- \leq x \leq q^+$
$\Rightarrow q^- \leq x \leq q^+$
$\Rightarrow q^- \leq y \leq q^+ - q_x$
NE Right Triangle Query

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:
\[ \mathcal{O}(n \log^2 n) \]
Query time:
\[ \mathcal{O}(1) \]

- 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
Can we do better?

### 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

### Recap:
- **kd-Tree**: General-purpose data structure for pts in $\mathbb{R}^d$
  - Orthogonal range query:
    - Count/report pts in axis-aligned rect.
    - $\text{Ans} = 4$
  - kd-Tree: Counting: $O(n)$ time
    - Report: $O(k + \log n)$ 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
      - Age range: $a_o \leq \text{age} \leq a_h$
      - Weight range: $w_o \leq \text{weight} \leq w_h$
    - Design a data structure for each criterion individually
    - Layer these structures together to answer full query

### Multi-Layer Data Structures

### 1-Dim Range Tree:
- **Goal**: Express answer as disjoint union of subsets
- **Method**: Search for $Q_{i_0} + Q_{i_1} + \ldots + Q_{i_k}$
- Assume extended tree
- Each node $p$ stores no. of entries in subtree: $p.size$

### Canonical Subsets:
- **Goal**: Express full answer as disjoint union of subsets
- **Method**: Search for $Q_{i_0} + Q_{i_1} + \ldots + Q_{i_k}$
- Assume extended tree
- Each node stores no. of entries in subtree: $p.size$
Recursive helper:
\[
\text{int range1Dx(Node p, Intv Q=[Q_L, Q_R], Intv C=[x_L, x_R])}
\]

initial call: \(\text{range1Dx(root, Q, C_0)}\)

Cases:
- \(p\) is external:
  - if \(p.pt.x \in Q\) \(\rightarrow 1\) else \(\rightarrow 0\)
- \(p\) is internal:
  - \(C \subseteq Q\) \(\Rightarrow\) all of \(p\)'s pts lie within query
  \(\rightarrow\) return \(p\).size
  \[x_L \leq p.C \leq x_R\]
  \(Q \cap C = \emptyset\) \(\Rightarrow\) none of \(p\)'s pts lie in \(Q\)
  \(\rightarrow\) return \(0\)
- Else partial overlap
  \(\rightarrow\) Recurse on \(p\)'s children
  + trim the cell

More details:
Given a 1D range tree \(T\):
- Let \(Q=[Q_L, Q_R]\) be query interval
- For each node \(p\), define interval cell \(C=[x_L, x_R]\)
  s.t. all pts of \(p\)'s subtree lie in \(C\)
- Root cell: \(C_0=[-\infty, +\infty]\)

Range Trees II

2D Range Searching:
- Layer a range tree for \(x\) with range tree for \(y\)
- For each node \(p \in 1D \times\) tree, let \(S(p) = \) set of pts in \(p\)'s subtree
- Def: \(p_{aux}\): A 1D-\(y\) tree for \(S(p)\)

Analysis:

Lemma: Given a 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 \(O(\log n)\) \(\rightarrow k\) to report
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

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

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

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

**Analysis:**
- The 1D x search takes $O(\log n)$ time + generates $O(\log n)$ calls to 1Dy search
  $\Rightarrow \text{Total: } O(\log n \cdot \log n) = O(\log^2 n)$

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

**2D Range Query Function**
```c
int range2D(Node p, Rect Q, Invr C = [x0, x1]) {
    if (p is external) return p.pt in Q? 1 : 0;
    else if (Q.x contains C) {    // C \subseteq Q.x projection
        [y0, y1] = [-\infty, +\infty]    // init y-cell
        return range1Dy(p.aux, Q, [y0, y1]);
    } else if (Q.x is disjoint of C) return 0
    else                      // partial x-overlap
        return range2D(p.left, Q, [x0, p.x])
        + range2D(p.right, Q, [p.x, x1]);
}
```
Hashing: (Unordered) dictionary
- stores key-value pairs in array table [0..m-1]
- supports basic dict. ops. (insert, delete, find) in \( O(1) \) expected time
- does not support ordered ops (getMin, findUp, ...)
- simple, practical, widely used

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

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 \( \text{Prob}(h(x)=h(y))=\frac{1}{m} \)
    - Eg. Let \( p \) - large prime, \( a \in [1..p-1] \)
    - \( h_{a,b}(x) = (ax + b) \mod p \mod m \)

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

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

Eg. Java variable names: \( \text{temp1, temp2, temp3} \)

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

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!
- Linear probing:
  - 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 $\lambda < \lambda_{\text{min}}$ or $\lambda > \lambda_{\text{max}}$? Rehash!
- Alloc. new table size $= \frac{n}{\lambda_0}$
- 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.

Hashing II

Example:

<table>
<thead>
<tr>
<th>Keys $(x)$</th>
<th>$h(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

$m = 8$

Analysis: Recall load factor $\lambda = \frac{n}{m}$
- $n = \#$ of keys
- $m = \text{table size}$

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

Thm: $S_{sc} = 1 + \frac{\lambda}{2}$
- $S_{sc}$ = Expected search time
- if $x$ found (successful)
- $U_{sc} = \text{Expected search time if}$
- $x$ not found (unsuccessful)

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

Designer: $\lambda_{\text{min}}, \lambda_{\text{max}}$ - allowed
- $\lambda$ value
- $\lambda_0 = \frac{\lambda_{\text{min}} + \lambda_{\text{max}}}{2}$

Proof: On avg. each list has $\frac{n}{m} = \lambda$
- Success: 1 for head + half the list
- Unsuccess: 1 " " + all the list

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

**Linear Probing:**
- $h(x), h(x)+1, h(x)+2, ...$
- Simple, but is it good?
- Let $S_{LP}$ = expected time for successful search.
  - $U_{LP} =$ "unsuccessful"

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

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

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

**Pf:** See latex notes.

**Hashing III**

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

**Quadratic Probing:**
- $h(x), h(x)+1, h(x)+4, h(x)+9, ...$

**Obs:** As $\lambda \to 1$ times increase rapidly.
Double Hashing:
(Best of the open-addressing methods)
- Probe sequence det'd by second hash fn.: g(x)
  \( h(x) + \{0, g(x), 2g(x), 3g(x)\ldots \mod m \} \)

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

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

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: 
- Proof is nontrivial (skip)

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

\( \lambda: 0.5 \quad 0.75 \quad 0.95 \quad 0.99 \)

\( S_{DH} \quad 1.39 \quad 1.89 \quad 3.15 \quad 4.65 \)

\( U_{DH} \quad 2 \quad 4 \quad 20 \quad 100 \)

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 \((a)\):
- delete \((a)\):
- find \((a)\):

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

Find \((x)\): Find entries on probe sequence until:
- found \(x\) \( \Rightarrow \) return \(v\)
- hit empty \( \Rightarrow \) return null
Multiway Search Trees:
- $a_1, a_2, a_3$
  - $x < a_1, a_2 < x < a_3, x > a_3$

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

B-Tree: of order $m$ ($\geq 3$)
- Root is leaf or has $\geq 2$ children
- Non-root nodes have $\lceil m/2 \rceil$ to $m$ children [null for leaves]
- $k$ children $\Rightarrow$ $k-1$ key-values
- All leaves at same level

Secondary Memory:
- Most large data structures reside on disk storage
- Organized in blocks - pages
- Latency: High start-up time
- Want to minimize no. of blocks accessed

Node Structure:
- Constant int $m$
- Class BTreeNode:
  ```cpp
  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 \([m/2] - 1\)
- Does either immediate sibling have extra? \(\geq [m/2] + 1\)
- Adopt child from sibling & rotate keys
- When applicable - preferred

Node Splitting:
- After insertion, a node has too many children ... \(m + 1\)
- We split into two nodes of sizes \(m' = \lfloor m/2 \rfloor\) and \(m'' = m + 1 - \lfloor m/2 \rfloor\)

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

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

B-Trees II

\(m = 5\)

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

Lemma: For all \(m \geq 2\),
\[\lfloor m/2 \rfloor \leq 2\lfloor m/2 \rfloor - 1 \leq m\]
\(\Rightarrow\) 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

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

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

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

Example:

How to save space?
- Store 1 char. per node
- $x$ \not\in $\Sigma'$ \Rightarrow try next char in $\Sigma'$
  \Rightarrow advance to next character of search string
- First-child/next-sibling

Tries and Digital Search Trees I

Analysis:
- 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
- \text{PATRICIA} = \text{Practical Alg. to Retrieve Info. Coded in Alpha...}
- Late 1960's: Morrison & Guschenerberg
- Each node has index field, indicates which char to check next (Increase with depth)

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

Path compression!

Example:
- \$S_1\$ is shortest prefix of \$S_i\$ unique to this string
- Eq: \(\text{ID}(S_i) = \text{"ama"}\)

Tries and Digital Search Trees II

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

Suffix Trees:
- Given single large text \(S\)
- Substring queries: "How many occurrences of "tree" in CMSC 420 notes"
- Notation: \(S^i = a_{a_1}a_{a_2}...a_{a_{i-1}}\)
- Suffix: \(S_i = a_{a_1}a_{a_2}...a_{a_{i-1}}\)
- Q: What is minimum substring needed to identify suffix \(S_i\)?
Example: $S = \text{pumapajama}$

Tries and Digital Search Trees III

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
      no. of external nodes in this subtree
    - else (fall out at external node)
      - compare target with string
        - if matches - found 1 occurrence
        - else - no occurrences

Suffix Trees (cont.)
$S$ - text string $|S| = n$
$s_i$ - 1st suffix
Substring ID = min subtr. needed to identify $s_i$
A suffix tree is a Patricia trie of the n+1 substring identifiers

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

Example:
Search ("ama") → End at internal node $r$
Report: 2 occ. → Goto $s_1$, verify
Search ("amapaj") → End at external node $i$

Claim:
This is a trie!

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:
  $x = 0.a_0a_1a_2...$, $a_i \in \{0, 1\}$
  $y = \mathcal{E}_y a_i \cdot \frac{1}{2^i}$

Example:

$$
\begin{align*}
\text{Point} & \quad x & \quad y \\
0 & \quad 0.0 & \quad 0.0 \\
1 & \quad 0.1 & \quad 0.1 \\
\end{align*}
$$

How do we extend to 2-D?

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

**Further Remarks:**
- Techniques for efficiently encoding, building, serializing, compressing...
- Can generalize to any dimension $x = 0.a_0a_1...$,
  $y = 0.b_0b_1...$  \( \mathcal{E}_n = a_0a_1b_0b_1... \)

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

**Bit Interleaving:**
Given a point $p = (x, y)$
$0 \leq x, y < 1$
let $x = 0.a_0a_1...$ in binary
$y = 0.b_0b_1...$
Define:
$\phi(x, y) = a_0b_0a_1b_1...$
Called Morton Code of $p$

**Proof:** By induction on no. of bits
Let $x = 0.a_0a_1...$  $y = 0.b_0b_1...$
and consider just $\phi(x, y) = a_0b_0...$

Define:
$\phi(x, y) = a_0b_0a_1b_1...$
Called Morton Code of $p$

The PR kd-tree + binary trie assign pts to same subtrees...
Deallocation Models:
- Explicit: (C++/C)
  - New/Dispose/Free
  - May result in leaks, if not careful
- Implicit: (Java, Python)
  - Runtime system deletes
  - Garbage collection
  - Slower runtime
  - Better memory compaction

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

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

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

Block Structure:
- Allocated:
  - InUse, prevInUse
  - 1
  - size

- Available:
  - InUse, prevInUse
  - 0
  - size

Memory Management

Guide:
- prevInUse: 1 if prev contiguous block is allocated
- prev/next: Links in avail list
- size/size2: Total block size (includes headers)

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

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 and avoids small fragments
Example: Alloc b=59

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 → add this to avail
- Else: merge with either/both to make max. avail block

Example:

Memory Management II

Some C-style pointer notation

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

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

Example: alloc(4) → alloc(10)

Allocation: alloc(b)
- $k = \lceil \log(b+1) \rceil$ add 1 for header
- if avail[k] non empty return entry + delete
- else: find avail[j] ≠ ∅ for $j > k$
- split this block

Big Picture:
- avail list is organized by level: avail[k]
- Block header structure same as before except:
  - prevInUse ≠ not needed

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

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

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