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 operations performed?

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

**Common:**
- \( O(1) \): constant time 😊
  - [Hash map]
- \( O(\log n) \): log-time (good)
  - [Binary search]
- \( O(n^p) \): \( p \): constant; poly time
  - \( O(n^3) \)

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

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

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

**Our approach:**
- **Theoretical**: Algorithms + Asymptotic Analysis
- **Practical**: Implementation + practical efficiency
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 (moving others back)
- \text{delete}(i): deletes \( i \)th item (moving others up)
- \text{length}(): returns num. of items

Implementations:
- Sequential: Store items in an array
- Linked allocation: linked list
  - Singly: head \( \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \) null
  - Doubly: head \( \rightarrow a_1 \leftarrow a_2 \leftarrow \ldots \leftarrow a_n \leftarrow \) tail

Performance varies with implementation

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
  - (top) - push + pop
  - push \[ \] \[ \] \[ \] \[ \] \[ \]
  - pop \[ \] \[ \] \[ \] \[ \] \[ \]
  - null

Queue:
- FIFO list: enqueue inserts at tail and dequeue deletes from head
  - \text{enqueue}: \[ \] \[ \] \[ \] \[ \] \[ \]
  - \text{dequeue}: \[ \] \[ \] \[ \] \[ \] \[ \]
  - \text{head} \[ \] \[ \] \[ \] \[ \] \[ \]
  - \text{tail} \[ \] \[ \] \[ \] \[ \] \[ \]

Cost model (Actual cost)

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

is reallocated to size $2n$

Dynamic (Sequential) Allocation

- When we overflow, double

Eg. Stack

\[
\begin{array}{c|c|c}
\text{Actual} & \text{exp} & \text{actual cost} \\
\hline
9 & 3 & 7 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{new} & \text{new} & \text{new cost} \\
\hline
11 & 11 & +231 \\
\hline
\end{array}
\]

Basic Data Structures II

- Amortized analysis of dynamic stack

Proof:

- Break the full sequence after each reallocation → run

\[
\begin{array}{ccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\hline
\end{array}
\]

- 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!

Amortized Cost:

- Starting from an
empty stack, suppose that any
sequence of $m$ ops takes time $T(m)$.
- The amortized cost is $T(m)/m$.

Thm:

- Starting from an empty stack,
the amortized cost of our stack
operations is at most 5.
- [i.e. any seq. of $m$ ops has cost $\leq 5m$]

Charging Argument:

- Each request of push/pop we charge user 5 work tokens
- We use 1 token to pay for the operation + put other
4 in bank account.
- Will show there is enough
in bank account to pay actual costs.
**Fixed Increment**: Increase by a fixed constant
\[ n \rightarrow n + 100 \]

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

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

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

Leave as exercise (spoiler alert!)
- **Fixed increment** → no
- **Fixed factor** → yes
- **Squaring** → yes

**Dynamic Stack**:
- Showed doubling \( \Rightarrow \) Amortized \( \mathcal{O}(1) \)
- 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) \( \mathcal{O}(n \cdot m) \) space
- Sparse matrix: Most entries are zero
Graph: $G = (V, E)$
- $V$: finite set of vertices (nodes)
- $E$: set of edges (pairs of vertices)

Depth: path length from root

Height: (of tree) max depth

Degree (of node): number of children

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

Rooted tree: A free tree with root node

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

"Family" Relations:
- Grandparent
- Parent
- Siblings
- Child
- Grandchild
- Leaf: no children
Representing rooted trees:
Each node stores a (linked) list of its children

Node structure:
- data
- firstChild
- nextSibling

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;
    ...
}
```

In Java:
```
root: a
  ... one more node
```

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

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

Binary Tree Traversals (can be generalized)

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

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)

Observation: Every extended binary tree is full

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

Those wasteful null links....

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 → inorder predecessor
Null right → "successor"
Dictionary:
- **insert** (Key x, Value v):
  - Insert (x, v) in dict. (No duplicates)
- **delete** (Key x):
  - Delete x from dict. (Error if x not there)
- **find** (Key x):
  - Returns a reference to associated value v, or null if not there.

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

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

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

Can we achieve O(log n) time for all ops? Binary Search Trees
- Idea: Store entries in binary tree sorted (inorder traversal) by key
- Example: find(s)

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

Replacement Node?
Inorder successor
Inorder predecessor

BSTNode insert(Key x, Value v, BSTNode p) {
    if (p == null)
        p = new BSTNode(x, v);
    else if (x < p.key)
        p.left = insert(x, v, p.left);
    else if (x > p.key)
        p.right = insert(x, v, p.right);
    else throw exception ⇒ Duplicate!
    return p;
}

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

Why did we do: p.left = insert(x, v, p.left)?
Sure you understand this.
BSTNode delete (Key k, 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 right null)
    if (p.left == null)
        return p.right
    if (p.right == null)
        return p.left
    r = findReplacement (p)
    copy r's contents to p
    p.right = delete (r.key, p.right)
return p

Example:

Find Replacement Node
BSTNode findReplacement (BSTNode p)
BSTNode r = p.right
while (r.left != null)
    r = r.left
return r

Example:

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
Balance factor: 
\[ \text{bal}(v) = \text{hgt}(v.\text{right}) - \text{hgt}(v.\text{left}) \]

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

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

AVL Trees I
- Basic defs
- Height props
- Rotations

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

Proof: (Induct. on \( h \))
\[
\begin{align*}
\text{base:} & \quad h = 0 \quad n(h) = 1 = F_0 - 1 \\
\text{Induction:} & \quad h = 1 \quad n(h) = 2 = F_1 - 1 \\
\text{Ind. Hyp.:} & \quad n(h) = 1 + n(h-1) + n(h-2) \\
& = 1 + (F_{h-2} - 1) + (F_{h-1} - 1) \\
& = (F_{h-2} + F_{h-1}) - 1 = F_{h+1} - 1
\end{align*}
\]

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

Proof: Fact: \( F_n \approx \varphi^n / \sqrt{5} \), where \( \varphi = (1 + \sqrt{5})/2 \) "Golden ratio"
\[
\begin{align*}
\varphi^h &= c \cdot \varphi^h \\
\Rightarrow h \leq \log \varphi n + c' \\
&= O(\log n)
\end{align*}
\]

BSTNode rotateRight(BSTNode p){
  BSTNode q = p.left
  p.left = q.right
  q.right = p
  return q
}
AVL Node 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 ...
    p = rotateLeftRight(p)
updateHeight(p); return p

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

AVL Trees II
- double rotations
- insertion

Find: Same as BST.
Insert: Same as BST but as we "back out" rebalance
How to rebalance? Bal = -2
Left-right heavy:

Utilities:
int height(AVLNode p)
return {p == null \to -1, \text{otherwise} \to p.height}

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

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

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

AVL Tree:
AVLNode: Same as BSTNode but
+ member int height

Bestrode rotate(left-right) (BSTNode p)

right-left rotation:

Double rotations:
left-right rotation:

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

AVL Trees III
- Deletion
- Examples

Examples:

AVL Node delete (Key x, AVLNode p)

Same as BST delete

Return rebalance(p)
Node types:

- **2-Node**
  - 1 key
  - 2 children
  - Example: 2-3 tree of height 2

- **3-Node**
  - 2 keys
  - 3 children

Recap:

- **AVL**: Height balanced
- **Binary**: 2-3 tree: Height exact
- **Variable Width**: Identical heights

Definition:

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

Adoption (Key Rotation):

1 + 3 = 2 + 2

Merge:

1 + 2 / 2 + 1 → 3

Split:

4 → 2 + 2

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

Roughly: \log_3 n ≤ h ≤ \log_2 n

How to maintain balance?

- Split
- Merge
- Adoption (Key rotation)

Conceptual tool:

We'll allow 1-nodes + 4-nodes temporary
**Dictionary operations:**

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

**Insertion example:**
```
insert(6)
```

**Delete Example:**
```
delete(5)
```

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

**Example (continued):**
```
merge
```
```
adopt
```
Encoding 3-node as binary tree node

![Diagram showing encoding of 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. q is red ⇒ q.level == p.level

**What we need are stricter rules!**

**AA-tree**:

- Arne Anderson 1993
- New rule:

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

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

  ![Diagram of skew operation]

  How to test?  
  \[ p\.left\.level = p\.level \]

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

  ![Diagram of split operation]

  How to test? (not needed, levels are monotone)
  \[ p\.level = p\.right\.level = p\.right\.right\.level \]

Example:

- **2-3 Tree**: 

  ![Diagram of 2-3 tree]

- **AA tree**: 

  ![Diagram of AA tree]

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

**AA Node split (AA Node p)**

```plaintext```
if (p == nil) return p
if (p.right.right.level == p.level) {
    AANode q = p.right
    p.right = q.left
    q.left = p
    q.level += 1 ← move q up a level
    return q ← new subtree root
} else return p ← everything's fine
```
```
Example:

```
int insert(AANode p, int x, int v)
{
    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:
  - `updateLevel`: If p's level exceeds \( l = 1 + \min(p_.left_.level, p_.right_.level) \)
  - `fixAfterDelete`: Same as AVL deletion, but end with:

  ```
  return fixAfterDelete(p);
  ```
History:
1989: Seidel & Aragon
[Explosion of randomized algorithms]
Later discovered this was already known: Priority Search Trees from different context (geometry) McCreight 1980

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

Geometric Interpretation:
key \rightarrow x
priority \rightarrow y

Example:

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

Along any path - Insertion times increase

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

? Is it always possible to do both?
Yes: Just consider the corresponding BST.
Insertion: As usual, find the leaf and create a new leaf node.
- Assign random priority
- On backing out - check heap order + rotate to fix.

Example:

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

Proof: Follows directly from BST analysis

Implementation: (see pdf notes)

Node: Stores priority + usual...

Helpers:
- lowest priority (p) returns node of lowest priority among:
- restructure: performs rotation p.left (if needed) to put lowest priority node at p.

Example:

Deletion: (cute solution) Find node to delete. Set its priority to $+\infty$.
Rotate it down to leaf level + unlink.
Idea: Skip list:
- Organize list in levels
  - Level 0: Everything
  - Level 1: Every other
  - Level 2: Every fourth
  - Level 3: Every 2^n

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
}
```

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
}

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 $\leq 1/n$.

\[ \rightarrow \text{Prob that any given node's level exceeds } l \text{ is } 1/2^l \quad \text{[l consecutive heads]} \]

\[ \rightarrow \text{Prob that any of n node's level exceeds } l \text{ is } \leq n/2^l \quad \text{[n trials with prob } 1/2^l] \]

\[ \rightarrow \text{Let } l = c \cdot \log n \quad \text{[log base 2]} \]

\[ \rightarrow \text{Prob that max level exceeds } c \cdot \log n \text{ is: } \]

\[ \leq n/2^l = n/2^c \quad \text{[using } l = c \cdot \log n] \]

\[ = n/2^c = 1/n^c \]

**Obs:** Prob. level exceeds $3 \cdot \log n$ is $\leq 1/n^2$.

(If $n \geq 1,000$, chances are less than 1 in million!)

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

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

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

\[ \Rightarrow E(n_i) = n/2^i \]

Total space (expected) is:

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

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

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

**Proof:**

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

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

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

**Cases:**

- Let $E(2^n, m_i) = E(i) = 1 + \text{(Prob[A])} E(i) + \text{(Prob[B])} E(i-1)$

Current node $\rightarrow$ same level $\rightarrow$ from prior level

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

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

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

**Basis:** $E(0) = 0 \Rightarrow E(1) = 2 \cdot \log n$

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

$\Rightarrow$ We visit $2 \cdot n$ 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)

**Delete (12)**

**Insert (24)**

**Analysis:** All operations run in time ~ \( \Theta(\log n) \) expected

**Note:** Variation in running times due to randomness only—not sequential

\( \Rightarrow \) 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

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

Tree's height has reduced by ~ half!
ZigZig(p): [LL case]

Subtrees A, C move up↑

ZigZig(p): [LR case]

Subtrees C, E of p move up↑

Zig(p): [L case]

Subtree A moves up↑ C unchanged

Splay (Key x):
Node p ← find (x) [nearest node]
while (p ≠ root) {
    if (p is child of root) zig(p)
    else /* p has grand parent */
        if (p is LL or RR grand child) zigZig(p)
        else /* p is LR or RL gr. child */ zigZag(p)
    else /* p is root */
        return p
}

insert (x):
splay (x)
qu = new Node (x)
if (root. key < x)
x. left = root
x. right = root. right
root. right = null
else /* symmetrical */
**Splay Trees**

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

**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.

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

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

**Dynamic Finger Theorem:**
- Keys: $x_1, \ldots, x_n$. We perform accesses $x_i, x_{i+1}, \ldots, x_n$.
- Let $\Delta_j = |x_j - v_{j-1}|$, distance between consecutive items
- $x_n$.
- Thm: Total access time is $O(m+ n \log n + \sum_{j=1}^{n-1} (1 + \log \Delta_j))$
Multiway Search Trees:
- A tree with many branches and levels
- Each node can have multiple children

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

Node Structure:
- Constant int M:
  - Class BTreeNode:
    - int nChild // no. of children
    - BTreeNode child[M] // children
    - Key key[M-1] // keys
    - Value value[M-1] // values

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 \( \lceil \frac{m}{2} \rceil \) to m children [null for leaves]
- k children ⇒ k-1 key-values
- All leaves at same level

Example: m=5
- Each node has 3-5 children
- 2-4 keys

Theorem: A B-tree of order m with n keys has height at most \( \frac{\lg n}{\gamma} \), where \( \gamma = \lg \left( \frac{m}{2} \right) \)
(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

B-Tree restructuring:
- Generalizes 2-3 restructuring
- 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 \(m' = [m/2]\) and \(m'' = m+1-\lceil m/2 \rceil\)

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

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

B-Trees II

Lemma: For all \(m \geq 2\),
\([m/2] \leq 2[m/2]-1 \leq m\)
\(\Rightarrow\) Resulting node is valid
**Insertion:**
- Find insertion point (leaf level)
- Add key/value here
- If node overfull (m keys, m+1 children)
  → Can either sibling take a child (<m)?
  → Key rotation [done]
  → Else, split
    → Promotes key
    → If root splits
      add new root

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

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

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

**B-Trees III**
**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:**
```
Example:

$\begin{array}{cccc}
 & & & 5 \\
 & & d & \\
 & b & & e \\
 a & & c & f \\
\end{array}$
```

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

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

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

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

**Insert:** $n++, m++$

**Delete:** $n--$
  - If $m > 2n$ rebuild

**How to rebuild?**
- Inorder traverse $p$'s subtree -> array $A[]$
  - 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($x, L, R$)
Details of Operations:

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

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

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

**Example:**

- **Init:** \( n \leftarrow m \); \( \text{root} \leftarrow \text{null} \)

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

**Scapegoat Trees II**

- *Must there be a scapegoat? Yes!*

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

**Proof:** By contradiction

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

\[
\text{depth of } u, \text{size}(u \text{'s child}) \leq \frac{2}{3} \cdot \text{size}(u)
\]

- **Inc.** \( n \):
  - 1 node:
    - \( 1 \leq \text{size}(p) \leq \frac{2}{3} n \)
    - \( \frac{3}{2} n \leq \text{size}(p) \leq \frac{2}{3} n \)
    - \( d \geq \log_{3/2} n \)

- **Dec.** \( n \):
  - \( d > \log_{3/2} n \)
  - \( \frac{3}{2} n \leq \text{size}(p) \leq \frac{2}{3} n \)
  - \( d < \log_{3/2} n \)

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

Proof: (Sketch)

- **Find:** $O(\log n)$ guaranteed [Height=$O(\log n)$]
- **Delete:** In order to induce a rebuild, number of deletes $\sim$ number of nodes in tree
  - Amortize rebuild time against delete ops
- **Insert:** Based on potential argument
  - It takes $\sim k$ ops to cause a subtree to size $k$ to be unbalanced.
  - Charge rebuild time to these operations
Weight Balance:
- Given a set of keys
  \( X = \{ x_0, \ldots, x_{n-1} \} \)
- and values
  \( V = \{ v_0, \ldots, v_{n-1} \} \)
- and weights
  \( W = \{ w_0, \ldots, w_{n-1} \} \)
- Assume:
  \( x_0 < x_1 < \ldots < x_{n-1} \) sorted
  \( w_i > 0 \) positivity

Pseudo-Probability:
- Let: \( W = \sum_{i=0}^{n-1} w_i \) total weight
- Let: \( p_i = \frac{w_i}{W} \) pseudo-prob
- Obs: \( 0 < p_i < 1 \) \( \Rightarrow \) discrete
  \( \sum_i p_i = 1 \) \( \Rightarrow \) prob. distribution

Shannon's Theorem: If \( p_i \) is the prob. of accessing \( x_i \), any BST has expected search at least \( \sum_i p_i \log_2 p_i \) \( \Rightarrow \) (called the entropy of distrib)

Overview:
- Splay trees - Static
  Optimal
- More frequently accessed keys closer to root
  \( \Rightarrow \) Weight-balanced
  trees

Implementation: (as extended BST)

Internal node: Stores:
- Key key \( \rightarrow \) splitter
- float wt \( \rightarrow \) total weight of entries in subtree

External Node:
- Key key, \( \leftarrow x_i \)
- Value value \( \leftarrow w_i \)

Given \( \frac{1}{2} \leq \alpha \leq 1 \), a BST is \( \alpha \)-balanced if for all internal nodes \( p \),
  \( \text{balance}(p) \leq \alpha \)

\( \alpha = \frac{1}{2} \): Perfectly balanced
\( \alpha = 1 \): Arbitrarily bad
\( \alpha = \frac{2}{3} \): A reasonable compromise

Weight-Balanced
Trees

How to (Nearly) Achieve
Shannon's bound
\( \Rightarrow \) Weight-balanced
tree
\( \Rightarrow \) For each node \( p \):
  \( \text{wt}(p) = \text{total weight of keys in } p \) subtree
  \( \text{balance}(p) = \max(\text{wt}(p.\text{left}), \text{wt}(p.\text{right})) / \text{wt}(p) \)
Balance by Rebuilding:
Given an array $A[0..k-1]$ of external nodes:
- Assume keys are sorted
- Assume weights $> 0$

**Weight-based median:**
- Select splitter to minimize left-right weight difference
- Assume keys are sorted
- Assume weights $> 0$

**Options:**
- **Rotations:** Similar to AVL trees (single + double)
  --- $\Rightarrow$ BB$[\alpha]$ trees
- **Rebuild subtrees:** Similar to scapegoat

**Example:**
Given an array $A[0..2]$

$$ \Delta_{\min} = \left| (1+2+4) - (3+2+4) \right| $$

$$ \bar{w} = 6 $$

**Weight Balanced Trees II**

**buildTree ($A[0..k-1]$):**
if ($k = 1$) return $A[0]$ / base case */

$$ \bar{w} = \sum_{i=0}^{k-1} A[i].w $$ / total weight */

Init: $b = 0$; $Lut = 0$; $Rut = \bar{w}$; $\Delta_{\min} = \bar{w}$
for ($i = 0 \ldots k-1$)
  $\bar{Lut} += A[i].w$; $\bar{Rut} -= A[i].w$

$\Delta = | \bar{Rut} - \bar{Lut} |$ / weight difference */
if ($\Delta < \Delta_{\min}$) \{ $b = i+1$; $\Delta_{\min} = \Delta$ \}

$L = \text{buildTree} (A[0..b-1])$ $R = \text{buildTree} (A[b..k-1])$
return new Int Node($A[b].key$, $L$, $R$)
But it is pretty close! 😊

**Theorem:** (Mehlhorn '77)
The above balanced split algorithm produces a tree whose exp. search time is
\[ \leq H + 3 \]
where \( H \) = entropy bound.

**Dictionary Operations:**
- **Balance by destroying & rebuilding:** Jackhammer Trees
- **Find:** Same as usual. Tree height \( \leq \log_3 n \), so \( O(\log n) \) time guaranteed.
- **Insert/Delete:** Start same as standard BST
  - After operation completes check & rebuild

**Analysis:**
Does this algorithm produce the optimal tree (w.r.t. expected case search time)?
- No. 😞 The optimal BST can be computed by dynamic programming
  - \( \text{Weight-Balanced Trees III} \)
  - \( \text{CMSC 451} \)

- **Check & Rebuild:**
  - When returning from recursive calls, update each node's weight
    \[ p.wt \leftarrow p.left.wt + p.right.wt \]
  - Starting at root, walk down search path. Stop at first node \( p \) s.t.
    \[ \text{balance}(p) > \alpha \]
    - Given by designer \( \alpha = 2/3 \)
    - Recall def earlier

**Bad weight distributions?**
- If a weight is very large relative to neighbors, rebalance may be ineffective

**Lemma:** If weights are "nice" (not too much variation), insert & delete run in \( O(\log n) \) amortized time.

- **If no such \( p \) found**
  - Great! Tree is balanced
- **Else:** Jackhammer!
  - Traverse \( p \)'s subtree in order, store extern nodes in array \( A[0..k] \)
  - Replace \( p \)'s subtree with
    \[ \text{buildTree}(A) \]
Very heavy entries:
- If an entry's weight is too high, rebuilding is ineffective
- Example:

```
    G /\ L
   / \   
  a   b   c
  1   8   1
```

- This tree is best possible!

- Exemption: Don't rebuild if a key's weight is very high

For node $p$: $\text{max}(p) = \text{max weight in } p\text{'s subtree}$

$\text{max-ratio}(p) = \frac{\text{max}(p)}{\text{weight}(p)}$

Given parameter $0 < \beta < 1$, a node is $\beta$-exempt if $\text{max-ratio}(p) > \beta$.

Dictionary Operations:
- find: as usual
- insert: insert as usual but rebuild if needed
- delete: delete as usual but rebuild if needed

Weight-Balanced Trees IV

When to rebuild?
- When "backing out" from insert/delete, update node weights
- Walk down search path from root [opposite from scapegoat!]

- If any node $p$ is out of balance:
  - $\text{balance}(p) > \alpha$
  - and
  - $\text{max-ratio}(p) \leq \beta$

  then:
  - Rebuild $p$:
    - Traverse $p$'s subtree inorder
    - Collect external nodes in array $A[0..k-1]$
    - replace $p$ with buildTree($A$)

Lemma: For any set of weighted entries, $\exists$ an ($\alpha, \beta$)-balanced $\text{BTree}$ if $\frac{1}{2} < \alpha < 1$ and $\beta < 2\alpha - 1$.
**Hashing**: (Unordered) dictionary
- stores key-value pairs in array table \([0..m-1]\)
- supports basic dict. ops. (insert, delete, find) in \(O(1)\) expected time
- does not support ordered ops (getMin, findUp, ...)
- simple, practical, widely used

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

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

**Overview**:
- To store \(n\) keys, our table should (ideally) be a bit larger (e.g., \(m > c \cdot n\), \(c=1.25\))
- Load factor:
  \(\lambda = \frac{n}{m}\)
- Running times increase as \(\lambda \rightarrow 1\)
- Hash function:
  
  \(h\): Keys \(\rightarrow [0..m-1]\)
  - 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:
- \(temp^1, temp^2, temp^3\)

**Common Examples**:
- Division hash:
  
  \(h(x) = (ax \mod p) \mod m\)
  
  a, p - large prime numbers
- Multiplicative hash:
  
  \(h(x) = (ax \mod p) \mod m\)
  
  a, b, p - large primes
- Linear hash:
  
  \(h(x) = ((ax+b) \mod p) \mod m\)
Overview:
- Separate Chaining
- Open Addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing

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

Separate Chaining:
- Table[i] is a linked list of keys that hash to i.

Insertion:
- Insert (x, v) → table[h(x)] = v

Finding:
- Find (x) → return table[h(x)]

Removal:
- Delete (x) → table[h(x)] = null

Thm: Amortized time for rehashing is 1 + (2λmax/(λmax - λmin))

How to control λ?
- Rehashing: If table is too dense/too sparse, reallocate to new table of ideal size.
- Designer: λmin, λmax - allowed λ value
  - λ₀ = (λmin + λmax) / 2 - "ideal"
  - If λ < λmin or λ > λmax ...

Analysis:
- Recall load factor λ = n/m
  - n = # of keys
  - m = table size

Proof:
- On avg, each list has n/m = λ
  - Success: 1 for head + half the list
  - Unsuccess: 1 "" "" + all the list
Open Addressing:
- Special entry ("empty") means this slot is unoccupied
- Assume $\lambda \leq 1$
- To insert key:
  check: $h(x)$ if not empty try
  $\langle h(x)+i_1, h(x)+i_2 \rangle$
- $\langle i_1, i_2, i_3, \ldots \rangle$ - Probe sequence
- What's the best probe sequence?

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

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

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

Let $S_{LP} = \frac{1}{2} (1 + \frac{1}{1-\lambda})$
$U_{LP} = \frac{1}{2} (1 + \frac{1}{1-\lambda})^2$

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

Quadratic Probing:
- $h(x)$, $h(x)+1$, $h(x)+2$, $h(x)+3$, ...

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

Obs: As $\lambda \rightarrow 1$ times increase rapidly

Linear Probing:
$h(x)$, $h(x)+1$, $h(x)+2$, ...

Simple, but is it good?
$x: d, z, p, w, t$

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

$\uparrow$ t did not collide directly but had to probe 3 times!

The table:
\[ \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array} \]
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:
- Quadratic probing:
- Clustering

Problem:
- $h(x):$ Apply find($x$)
  $\rightarrow$ Not found $\Rightarrow$ error
  $\rightarrow$ Found $\Rightarrow$ set to "empty"

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

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

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>$U_{DH}$</td>
<td>2</td>
<td>4</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>$S_{DH}$</td>
<td>1.39</td>
<td>1.89</td>
<td>3.15</td>
<td>4.65</td>
</tr>
</tbody>
</table>

Very efficient!
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 w. a region – cell
- Each internal node stores a splitter – subdivides the cell
- External nodes store pts.

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

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

Quadtrees & kd-Trees I

Quadrants:
- $\mathbb{R}^2$
- Lower-Left
- Upper-Left
- Lower-Right
- Upper-Right

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

Class Point{
  float[] coord // coords
  Point(int d)
  \ldots
  coord = new float[d]
  \ldots
  \ldots
}

int getDim() \rightarrow coord.length
float get(int i) \rightarrow coord[i]
\ldots
\ldots
others: equality, distance, toString...
**Point Quadtree:**
- Each internal node stores a point.
- Cell is split by horizontal and vertical lines through point.

**Quadtree (abstractly):**
- Partition trees
  - Cell: Axis-parallel rectangle
    - AABB: "Axis-aligned bounding box"
  - Splitter: Subdivides cell into four (generally 2d) subcells

**History:** Bentley 1975
- Called it 2-d tree ($\mathbb{R}^2$)
- 3-d tree ($\mathbb{R}^3$)
- In short $kd$-tree (any dim)
- Where/which direction to split? → next

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

**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)

**Quadtree Analysis:**
- Numerous variants!
  - PR, PMR, QR, QX, ... see Samet's book
- Popular in 2-d apps
  - (in 3-d, octrees)
- Don't scale to high dim
  - Out degree = $2^d$
- 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
}
```

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

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

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

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

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

Quadtrees & Kd-Trees III

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

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

**Kd-Tree Insertion:** (Similar to std. BSTs)

- Descend tree until cutting. 
  - if found: cutting node.
  - fall out: (Although we draw extended trees, let’s assume standard trees)

**Example:**

```
insert(3,4)
```

**Analysis:**

- **Run time:** $O(h)$

- **Can we balance the tree?**

- Rotation does not make sense

**Rebalance by Rebuilding:**

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

**Quadtrees & kd-Trees IV**

**Deletion:**

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

**This is the hardest part. See Latex notes.**
Kd-Trees:
- Partition trees
- Orthogonal split
- Alternate cutting
dimension \( x, y, x, y, \ldots \)
- Cells are axis-aligned
  rectangles (AABB)

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

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

\( \rightarrow \) General \( \mathbb{R}^d \): \( \mathcal{O}(n^{1-\frac{1}{d}}) \)

Rectangle methods for kd-cells:
- Split a cell \( r \) by a split
  pt \( s \in r \), along cutdim \( c \).
- \( r.h \leadsto r \), \( r.l \rightarrow \)
- \( r \rightarrow \) high
- \( r \rightarrow \) left part
- \( \text{leftPart}(cd, s) \)
  \( \rightarrow \) returns rect with \( \text{low}=r.lw \)
  \( \quad \text{high}=r.high \) \but high \( \leadsto s[cd] \)
- \( r \rightarrow \) right part
- \( \text{rightPart}(cd, s) \)
  \( \rightarrow \) high = \( r.high \) \+ \( \text{low}=r.lw \) \but
  low \( \leadsto s[cd] \)

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

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

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

Kd-Tree Queries II

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

int rangeCount(Rect R, KDNode p, Rect cell )
if (p == null) return 0 // fell out of tree
else if (R is Disjoint from (cell)) return 0 // overlap
else if (R.contains(cell)) return p.size // take all
else { int ct = 0
  if (R.contains(p.pt)) ct ++ // p's pt in range
  ct += rangeCount(R, p.left +
  cell.leftPart (p.cutDim, p.pt))
  ct += rangeCount(R, p.right, cell.rightPart...}
Theorem: Given a balanced Kd-tree storing $n$ pts in $\mathbb{R}^2$ (using alternating cut dim), orthog. range queries can be answered in $O(n^{1/2})$ time.

Analysis: How efficient is our algorithm?
- Tricky to analyze
- At some nodes we recurse on both children
  - $O(n)$ time?
- At some we don't recurse at all!

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

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

For us: $a = 2$, $d = 4$
$$T(n) = n^{\log_2 2} = n^{\log_b a} = n^{1/2} = \sqrt{n}$$

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

Reurrence: $T(n) = 2 + 2T\left(\frac{n}{4}\right)$
- 2 cells stabbed
- Each has $\frac{n}{4}$ pts
- Recurse on 2 grandchildren

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

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

Cases: $p$ splits
- horizontally
- vertically

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

Simpler: Extend $R$'s sides to 4 lines + analyze each one.
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 $n$ 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.
  - $\text{ans} = 4$
  - $\text{No. of pts reported} <$ 10
- **kd-Tree:** Counting: $O(n)$ time
  - Report: $O(k + \log^d n)$ time

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

**Layering:** Combining search structures
- Suppose you want to answer a composite query with multiple criteria:
  - Medical data: Count subjects
    - Age range: $a_{i0} \leq \text{age} \leq a_{in}$
    - Weight range: $w_{i0} \leq \text{weight} \leq w_{in}$
- Design a data structure for each criterion individually
- Layer these structures together to answer full query

**Call this a 1-Dim Range Tree:**

**1-Dim Range Tree:**
- Goal: Express answer as disjoint union of subsets
- Method: Search for $Q_{i0}$ and $Q_{ni}$ + take maximal subtrees

**Canonical Subsets:**
- Design a data structure for each criterion individually
- Layer these structures together to answer full query

**Multi-Layer Data Structures**
Recursive helper:

```
int range1Dx(Node p, 
    Intv Q=[Q_l, Q_h], Intv C=[x_o, x_i])
```

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

### Cases:

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

---

**More details:**

Given a 1-D range tree `T`: Let `Q=[Q_l, Q_h]` be query interval

- For each node `p`, define interval cell `C=[x_o, x_i]` s.t. all pts of p's subtree lie in `C`
- **Root cell:** `C_o=[-∞, +∞]`

**Range Trees II**

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

---

**2-D Range Searching:**

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

**Analysis:**

**Lemma:** Given a 1-D range tree with `n` pts, given any interval `Q`, can compute `O(\log n)` sub-trees 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

x-range tree

p.aux

\[\text{Range Trees III}\]

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

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

int range2D(Node p, Rect Q, Invtr C = [x_0, x_1])
if (p is external) return $p.p.t \in Q$
else if ($Q \times$ contains $C$)
  $\langle y_0, y_1 \rangle = [-\infty, +\infty]$ // init $y$-cell
  return range1Dy(p.aux, Q, $\langle y_0, y_1 \rangle$)
else if ($Q.x$ is disjoint of $C$) return 0
else
  return range2D(p.left, Q, $[x_0, p.x]$) + range2D(p.right, Q, $[p.x, x_1]$)

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

Intuition: The $x$-layer finds subtrees $p$ contained in $x$-range + each aux tree filters based on $y$.
Persistent Data Structures:
- Preserves prior states of data structure
- Allows searches in history
  → “Was Jane Smith enrolled in UMD in Spring 2016?”
- Full/Partial Persistence:
  - Change can be made to any/current version

Example: Rebuild subtree $T_1$ at time $t$. Let $T_2$ be new subtree:
- BJ Tree:
  - $T_1 \rightarrow T_2$
- PBJ Tree:
  - $T_1 \rightarrow T_2$

Approach: Whenever a modification made, save old contents & use temporal node to distinguish

Change at $t$:
- Old
- New

Example:
- Command:
  - 01: insert IAD
  - 03: insert BWI
  - 05: insert SFO
  - 07: delete BWI

Time:
- 01: IAD
- 03: BWI
- 05: SFO
- 07: IAD

Find IAD at 01: not found
Find IAD at 03: found
Get-min at 04: BWI
Get-min at 08: IAD

Case study: PBJ Tree
- Persistent Weight Balanced
- Jackhammer trees

→ A partially persistent BJ tree
→ Uses rebuilding to balance subtrees
→ Allows weighted entries

Persistent Search Trees

Temporality:
- New node type
- $t$ = time
- $t_0$ = post time
- $t_1$ = pre time
- $t_2$ = subtree contents
  - Prior to time $t$
  - On or after time $t$

Copy-on-write: Whenever you modify, make a copy
→ very inefficient!

Change-log: Make a list of recent updates
→ slow to process

Fat nodes/Node copying:
- When changes occur, save just modified portions

Temporal node: New node type.
- Prior to time $t$:
- On or after time $t$:

Approach:
- Whenever a modification made - save old contents & use temporal node to distinguish
- Full/Partial Persistence:
  - Change can be made to any/current version

Fat nodes:
- Node copying:
- When changes occur, save just modified portions

Driscoll, Sarnak, Sleator, Tarjan (1986) - First serious theoretical analysis
- Applied to geometric search (time is coordinate)

History:
- Nino, the theoretical analysis
- Allows searches in history
  → “Was Jane Smith enrolled in UMD in Spring 2016?”

Approaches:
- Copy-on-write: Whenever you modify, make a copy
  → very inefficient!
- Change-log: Make a list of recent updates
  → slow to process
- Fat nodes:
  - Node copying:
    - When changes occur, save just modified portions
  - Temporal node:
    - New node type

Time:
- 01: IAD
- 03: BWI
- 05: SFO
- 07: IAD

Find IAD at 01: not found
Find IAD at 03: found
Get-min at 04: BWI
Get-min at 08: IAD
PBJ Tree (Private) Data:
- final float ALPHA, BETA
  ≡ same as BJ Tree
- Node root - root → init: null
- int firstInsertTime } → init: -1
- int lastInsertTime

Insertion (without rebalancing).
Helper: Node insert(x, v, w, t)
→ insert(x, v) of wt w at t
- If root == null: (first insertion)
  [ firstInsertTime ← t
  root = new Ext Node (x, v, w)
- else
  search for x in tree
  - reach Ext Node y
    - if x == y → Error: Duplicate key
    - else:
      Parent of y
      - May be Int or Temp
      - [?] may be left or right (or post)
        child
      copy y's
      x < y
      x > y

Persistent Search Trees II

Few More Things:
- Before insert, check that
  \( t > \text{lastInsertTime} \)
→ else Error!
- Set lastInsertTime ← t
- Note: Partial persistence
- Rebalance → Next

Updating Weights?
Int Node: wt ← left. wt + right. wt
max wt ← max(left. max wt, right.)
Temp Node: wt ← post. wt ∧ ... only post!

Example:
Insert("D") time = 7
Node structure:
Node: weight + maxWt (float)
   \[ \text{TempNode: time (int)} \]
   \[ \text{pre, post (Node)} \]
   \[ \text{Int Node: key, left, right} \]
   \[ \text{Ext Node: key, value} \]
**Internal Node Rebalance:**
- Test α, β condition (same as BJ tree)
- If unbalanced, compile list $A$ of extern nodes for current tree
- $T'$ = buildTree($A$)
- return:

**Temporal Node Rebalance:**
- Recurse on post side: $post = post.rebalance(x, t)$
- On return:
  - if (post child is temporal)...
    - perform left rotation
    - update weights
  - return root of subtree

**Why?** Don't like long post chains

---

**Rebalancing after insertion:**
- Starting at root, retrace search path
- Recursive helper: Node rebalance($x, t$)
  - Internal: Apply to left/right based on $x \Rightarrow$ key
  - Temporal: Apply to post only.

**Persistent Search Trees**

**Example:**

**Find($x, t$), findUp($x, t$), getMin($t$):**
- Same as before but for temporal nodes visit pre/post based on $t$
- Check whether $t < \text{first Update Time}$
  - if so $\rightarrow$ null

**getPreorderList($t$)/getFullPreorderList()**
- First gets preorder list at time $t$ (no temporal nodes!)
- Second gets full preorder list (all nodes)

**Delete/Clear:** Not implemented

---

**Deep Copy?**
- We did deep
  - Does it matter?
**Tries:** History
- de la Briandais (1959)
- Fredkin: "trie from "retrieval"
- Pronounced like "try"

**Node:** Multiway of order k

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

**Search Trees I**

**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:**

```
root
  a
  /  \
 b   c
```

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

**Same structure/Alt. Drawing**

**Analysis:**
- Search: $\approx$ length of query string [O(1) time per node]
- Space:
  - No. of nodes $\approx$ total no. of chars in all strings
  - Space $\approx k \cdot$ (no. of nodes)
Patricia Tries:
- Improves trie by compressing degenerate paths
- PATRICIA = Practical Alg. to Retrieve Info. Coded in Alpha
- Late 1960's: Morrison & Gushienger
- 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: ID '$ - $ Improves trie by compressing degenerate paths
To get both good space & query time efficiency.

Tries and Digital Search Trees II

Same data structure - Drawn differently
- ski a. skipped: substring
discriminator: substring
- e

Essential:
- Essence: essential estimate sublime subliminal

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

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

Example: $S = \text{pamapajama}\$
- Def: Substring identifier for $S_i$ is shortest prefix of $S_i$ unique to this string
- $S_7 = \text{ama}\$
  - Eq: $\text{ID}(S_i) = \text{"ama"}$
  - $\text{ID}(S_7) = \text{"ama\"}$
Example: \( S = \text{pumapajama} \)

Suffix Trees (cont.)

- Text string \( |S| = n \)
- \( S_i \) = \( i^{th} \) suffix
- Substring ID = min substr. needed to identify \( S_i \)
- A suffix tree is a Patricia trie of the \( n+1 \) substring identifiers

Substring Queries:

- How many occurrences of \( t \) in text?
  - 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 of on external node)
    - compare target with string
    - if matches - found 1 occurrence
    - else - no occurrences

Example:

Search ("ama") \( \rightarrow \) End at internal node
  - Report: 2 occ.
Search ("amapaj") \( \rightarrow \) End at external node
  - Go to \( S_j \) + verify

Tries and Digital Search Trees III

Analysis:

- Space: \( O(n) \) nodes
  \( O(n \cdot k) \) total space
  \( (k = |S| = O(1)) \)
- Search time: \( n \) to length of target string
- Construction time: \( O(n \cdot k) \) [non-trivial]

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

Geometric Applications:

- PR k-d Tree: k-d tree based on midpoint subdivision
- Assume points lie in unit square

Example:

Geometric

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

Example:

\[
\begin{array}{c|c|c|c|c}
  & 0 & 0. & 0.0 & 0.1 \\
\hline
  0 & 1 & 0.0 & 0.1 & base 2 \\
  1 & 0 & x & y \\
\end{array}
\]

Bit string: 01010110001...

Tries and Digital Search Trees IV

Further Remarks:
- Techniques for efficiently encoding, building, serializing, compressing...
- Can generalize immediately to PR kd-tree

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

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

Define:
\[ \phi(x, y) = a_1b_1a_2b_2a_3b_3... \]
Called Morton Code of \( p \)

How do we extend to 2-D?

Given a point \( p = (x, y) \)
- Let: \( x = 0.a_1a_2... \) in binary
- \( y = 0.b_1b_2... \)

Define:
\[ \phi(x, y) = a_1b_1a_2b_2a_3b_3... \]
Called Morton Code of \( p \)

Example:

\[
\begin{array}{c|c|c|c|c}
  & 0 & 0. & 0.0 & 0.1 \\
\hline
  0 & 1 & 0.0 & 0.1 & base 2 \\
  1 & 0 & x & y \\
\end{array}
\]

Bit string: 01010110001...

How do we extend to 2-D?

Given a point \( p = (x, y) \)
- Let: \( x = 0.a_1a_2... \) in binary
- \( y = 0.b_1b_2... \)

Define:
\[ \phi(x, y) = a_1b_1a_2b_2a_3b_3... \]
Called Morton Code of \( p \)
Deallocation Models:
Explicit: (C++++)
- programmer deletes
- 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
- size

Available:
- inUse
- prevInUse

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

Fragmentation:
- results from repeated allocation/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

Internal: Induced by mem. manage. policies (not user)
External: Caused by pattern of alloc/dealloc

Internal:
- Induced by mem. manage. policies (not user)
- Faster + avoids small fragments

External: Caused by pattern of alloc/dealloc (Swiss-cheese effect)
Example: Alloc b=59

Allocation: `malloc(b)`
- Search avail. list for block of size `b' > 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

`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++  // 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
else {
    p.size -= b  // remove allocation
    *(p+p.size-1) = p.size  //size2
    q = p + p.size  //start of new block
    q.size = b
    q.prevInUse = 0  \new block header
    q.inUse = 1
    (q + q.size).prevInUse = 1
    // update prevInUse for next contig. block
    return q+1  \skip over header
}
```
**Buddy System:**
- Block sizes (including headers) are 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>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

**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:**

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

**Allocation:**
- k=\lceil \lg (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

**In practice:** There is a minimum allowed block size

Buddy system only allows allocations aligning with these blocks