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
- 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 $\mathbb{R}^d$ $p = (p_1, \ldots, p_d)$ $p_i \in \mathbb{R}$

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 $d$)
... $\rightarrow$ coord = new float[$d$]

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

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

Quadtrees & kd-Trees

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

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

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?
Announcements: 3/14 Happy Pi day!

- Midterm - Still grading
- Programming Assignment 2
  - Coming soon
  - Due Apr. 5
  - kd-trees ...

Gonzalez's Algorithm
- Repeatedly select the point farthest from any points already selected
Solution 1:
(a) AVL Tree: $\text{insert}(25)$

(b) AA Tree: $\text{insert}(24)$
Solution 2:

(a) You have a binary tree nodes with inorder threads. There are \( n \) nodes. **How many threads are there?** (If the answer cannot be determined from \( n \) alone, answer “It depends”).

**Answer:** \( n + 1 \): The threads replace the null child pointers in the binary tree. In class we showed that any binary tree with \( n \) nodes has \( n + 1 \) null child pointers.

(b) You have a union-find data structure for a set of \( n \) objects. After initialization, you perform \( k \) union operations, where \( 1 \leq k \leq n - 1 \). As a function of \( n \) and/or \( k \), what is the maximum possible height of any tree in the data structure?

**Answer:** \( \log(k + 1) \): Since the data structure starts with \( n \) individual singleton set, after \( k \) operations the largest any tree can be is \( k + 1 \) nodes (assuming that all \( k \) operations go to build the same tree). In class we showed that any tree in the union-find structure that has \( m \) nodes has height at most \( \log m \).
(c) We showed that in any leftist heap containing \( n \) nodes the NPL (null path length) of the root is \( O(\log n) \). What is the best that can be said about the NPL value of the root node in any binary tree having \( n \) nodes? (Select one.)

**Answer:** \( O(\log n) \): We showed that the NPL value of the root of any leftist tree is \( O(\log n) \). Any tree can be converted into a leftist tree by swapping the left and right children of nodes that violate the leftist property. Swapping child nodes does not alter the NPL values, so this bound holds for all binary tree, leftist or not.

(d) You perform the operation **update-key** in a leftist heap containing \( n \) entries. Assuming you do this by sifting the entry up or down the tree, what is the worst-case time for this operation? (Select one)

**Answer:** \( O(n) \): Even the rightmost path of a leftist heap is of \( O(\log n) \) length, this restriction does not apply to any other path. A path can be as long as \( O(n) \), and if you perform a sift-up that goes from the leaf to root level along such a path, the running time is \( O(n) \).
(e) You perform a postorder traversal of a binary tree with \( n \geq 1 \) nodes. True or false: The first node in the traversal must be a leaf.

**Answer:** True: A postorder traversal does not visit a node until after both its subtrees are visited, so the first node visited must have no children and hence is a leaf.

(f) You want to maintain an ordered dictionary using a simple linear list. List (very briefly) one advantage and one disadvantage of an array-based approach versus a linked-list approach.

**Answer:** Let's assume keys are sorted. **Advantage of arrays:** Faster search using binary search, less storage space (no need for pointers), better cache behavior for modern memory systems. **Disadvantage:** Need to reallocate periodically when space runs out. Also, insertion/deletion require shifting elements around.

\[
\begin{array}{ccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\circ & \circ & \circ & \circ & \circ & \circ & \circ \\
\end{array}
\]
(g) You have an AA tree whose root resides at level 13. What is the minimum and maximum number of red nodes that can be encountered along any path from the root to the leaf level (that is, level 1)?

Answer: Min: 0, Max: 13: You can visit one per level, but there need not be any red nodes in the tree at all.

A few people attempted to answer a much more challenging question. What is the maximum total number of red nodes in the tree? At each level of the tree, each black-red cluster gives rise to 3 more black-red clusters at the next level, so the number grows roughly as $\sum_{i=0}^{12} 3^i = (3^{13} - 1)/2$. This was a harder problem, so I gave credit if you got this answer.

(h) You have a data structure in which the $i$th operation takes time $O(i)$. Given a string of $n$ operations on this data structure, what is the best that can be said about the amortized time of each operation? Hint: $\sum_{i=1}^{m} i = m(m+1)/2$.

Answer: $O(n)$: The total time for $n$ operations is $T(n) = \sum_{i=1}^{n} i = O(n^2)$ and so the amortized time is $T(n)/n = O(n)$. 

\[ T(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \]

\[ T(n)/n = \frac{n}{2} \]
Solution 3(a):

Leftmost leaf of node p.

```java
Node leftLeaf(Node p) {
    if (p.firstChild == null) {
        return p; // p is a leaf?
    } else {
        return leftLeaf(p.firstChild) // get the left leaf of the left child
    }
}
```

Solution 3(b):

(b) Helper function computes the rightmost leaf reachable from `p` or any of its siblings. The answer is the rightmost leaf of `p.firstChild`.

```java
Node rightLeaf(Node p) {
    return rlHelper(p.firstChild); // get rightmost leaf in p’s subtree
}
```

```java
Node rlHelper(Node p) {
    if (p.nextSibling != null) { // can go right?
        return rlHelper(p.nextSibling); // go to the right
    } else if (p.firstChild != null) { // can go down?
        return rlHelper(p.firstChild); // go down
    } else { // no further?
        return p; // this is it
    }
}
```
Solution 4: Throughout, let $k$ denote the number of rows and columns in the matrix.

(a) Given $k \geq 1$, what is the total number of keys $n$ inserted? (Express your answer as a function of $k$.) **Hint:** For any $m \geq 0$, $\sum_{i=1}^{m} i = m(m+1)/2$.

**Answer:** $n = k(k+1)/2$: The total number of elements $n$ is obtained from adding up the number of elements in each column, which is $\sum_{i=1}^{k} i$, which by the standard formula is $k(k+1)/2$. Note that for large $n$ and $k$, we have $n \approx k^2/2$ or equivalently $k \approx \sqrt{2n}$.

(b) **Draw the final binary search tree** that results for the above example when $k = 5$. **Hint:** It has a very regular structure.
(c) What is the height of the resulting tree? (Express your answer as a function of \( k \) and/or \( n \).)

Answer: \( h = k - 1 \): All the leaves of the tree are at the same level, which is the side length of the matrix. Because heights start counting from zero, the tree’s height is \( k - 1 \). (This can be expressed as a function of \( n \), but it is quite messy. We get \( h = (\sqrt{1 + 8n - 3})/2 \approx \sqrt{2n.} \))

(d) Letting \( h \) denote the height of the final tree. Give a formula \( d(i) \), which for \( 0 \leq i \leq h \) is the number of nodes at depth \( i \) in the resulting tree. Express your answer as a function of \( k, n, \) and/or \( i \).

Answer: \( d(i) = i + 1 \): The number of nodes per level increases linearly from 1. Since depths start with zero, we have \( d(i) = i + 1 \).
(e) Assuming that the time needed to insert a node at depth \( i \) is \( i + 1 \), what is the total time needed to insert all \( n \) keys in the tree? For full credit, express your answer in big-O notation as an tight asymptotic function of \( n \) in closed form—no summations or recurrences. **Hint:** For any \( m \geq 0 \), \( \sum_{i=1}^{m} i^2 = (2m^3 + 3m^2 + m)/6 \).

**Answer:** \( T(n) = O(n^{3/2}) \): The total insertion time \( T(n) \) is the sum of the product \((i+1)d(i)\) for \( i \) ranging from zero up to the height of the tree. Thus, we have

\[
T(n) = 1 \cdot 1 + 2 \cdot 2 + \ldots + k \cdot k = \sum_{i=1}^{k} i^2.
\]

By the standard formula for the quadratic summation, this is \( (2k^3 + 3k^2 + k)/6 = O(k^3) \). Since asymptotically, \( k = O(\sqrt{n}) \), we have \( T(n) = O(n^{3/2}) \).
Solution 5: Expanding stack with sizes 1, 4, 9, 16, ..., $k^2$, ...

(a) Consider a single run involving an array of size $m = k^2$ for some large $k$. Derive the worst-case amortized cost of this single run.

Answer: $\approx \sqrt{\frac{m}{2}}$.

At the start of the run the array has $(k-1)^2$ elements. We overflow when we hit $k^2$ elements, so we can perform at least $k^2 - (k-1)^2 = 2k - 1$ operations until the next expansion. Let us charge the $\tau$ work tokens for each operation. We take one work token to pay for the actual operation, and we bank the remaining $\tau - 1$ tokens. When the next expansion occurs, we have saved at least $(2k-1)(\tau-1)$ tokens. The expansion cost is the number of elements copied, which is $m = k^2$. To pay for the expansion, we need to set $\tau$ large enough so that

$$(2k-1)(\tau-1) \geq k^2 \quad \text{or equivalently} \quad \tau \geq 1 + \frac{k^2}{2k-1} = \frac{k}{2} + \frac{\sqrt{m}}{2},$$

assuming that $k$ is large.
(b) We start with an empty stack and an array of size \( m = 1 \) and perform \( n \) operations, for some very large number \( n \). What is the worst-case asymptotic amortized cost of this data structure as a function of \( n \)?

**Answer:** \( O(\sqrt{n}) \). We break any sequence of operations of (large) length \( n \) into runs between expansion events. We have shown in (a) that the amortized cost of any run is \( O(\sqrt{m}) \) where \( m \) is the size of the current array. After \( n \) operations, the largest the array can be is \( n \) (assuming nothing but pushes), so the amortized cost cannot be larger than \( O(\sqrt{n}) \).

This is a good first-order estimate (and would be good enough for full credit). Here is a more detailed analysis. Since we’re looking for an asymptotic bound, there is no harm in rounding \( n \) up to the next larger perfect square. Let \( n = k^2 \). This means that we have gone through \( k \) runs to get here (with arrays of size 1, 4, 9, \ldots, \( k^2 \)). As argued in part (a), each run is dominated by the copying cost, which is the current size of the array. Thus, the cost of the \( i \)th run is roughly \( i^2 \). Applying the quadratic summation formula, we see that the total cost is therefore

\[
\sum_{i=1}^{k} i^2 = \frac{2k^3 + 3k^2 + k}{6} \approx \frac{k^3}{3}
\]

for large \( k \).

Since \( n = k^2 \), the total cost is roughly \( T(n) = n^{3/2}/3 \). To get the amortized cost, we divide by the number of operations \( n \), which yields an amortized cost of \( T(n)/n = \sqrt{n}/3 \), which is \( O(\sqrt{n}) \), exactly as given by our quick analysis.