DIFFERENT PURPOSES

General Purpose

Structure designed to facilitate fast lookup, addition, and removal based on a single key
Compete with Hash tables
Examples:
- Binary Search Tree
- Red Black Tree (java TreeMap)
- AVL Tree

Fast Access, self-optimizing

Attempts to optimize the tree such that recently used entries reside near the tree’s root
Example:
- Splay Tree
Uses: Network routers, garbage collection, caches, etc. …
DIFFERENT PURPOSES

Minimizing Size Overhead

Reduces the amount of overhead the tree requires as \( n \) grows

Examples:
- B-Tree
- 2-3 Tree

Spatial

Maps entries based upon their proximity to one another

Example:
- PR-Quadtree
- Point Quadtree
- Bounding Sphere Tree
- Binary Space Partitioning Tree
2-3 TREES
BINARY NODES

Observe that in a binary search tree, we need:

- 64-bit address for the entry
- 64-bit address for left child
- 64-bit address for right child

**Total:** 192 bits for every node

Thus, storing:

- 2 data entries requires: **384** bits
- 4 data entries requires: **768** bits
3-ARY NODES

By modifying the node to hold two values:

- 64-bit address for the left entry
- 64-bit address for the right entry
- 64-bit address for left child
- 64-bit address for the middle child
- 64-bit address for right child

**Total**: $64 \times 5 = 320$ bits for every node

Thus, storing:

- 2 data entries requires: **320** bits
- 4 data entries requires: **640** bits

128 fewer bits required to store 4 entries

Less overhead space
2-3 TREES

Self-balancing tree providing guaranteed average and worst case search, insertion, and deletion in $O(\log n)$

**Uses:**
- Data resides in some kind of external storage (e.g., hard drive, SSSD 8” floppy, Blu-ray, etc. . . .)
- Expensive to access

Higher data density means fewer reads from slow storage

Image source: http://cs.middlesex.cc.nj.us/~schatz/csc236/worksheets/2-3tree.worksheet.answers.html
GENERAL PROPERTIES OF 2-3 TREES

Tree Consists of:

- Internal binary nodes with two children
- Internal 3-ary nodes with three children
- Binary and 3-ary leaf nodes with zero children

A complete tree with all leaves at the same level
Data kept in sorted order

Observe: All internal nodes have all possible children (no null values)
Analogous to searching for an item in a standard binary search tree. Let $d$ be the datum we wish to find

1. Empty tree, so not present
2. The root is a leaf node, so $d$ is part of the tree only if it is one of the root’s entries
3. If root is a binary node
   1. Value $d$ equals the node’s entry
   2. If $d$ is less than the node’s entry, continue the search in the left subtree
   3. If $d$ is greater than the node’s entry, search in the right subtree
4. If root is a 3-ary node
   1. If $d$ is equal to one of the node’s entries, then we have found the item in the tree
   2. If $d$ is less than the 3-ary node’s entry, then continue the search in the left subtree
   3. If $d$ is greater than the node’s smaller entry and less than its larger entry, search in the middle subtree
   4. If $d$ is greater than the node’s larger entry, search in the right subtree
INSERTION IN A BINARY NODE

Initial

Final

Leaf binary node transforms into a 3-node
INSERTING IN A 3-NODE WITH A BINARY PARENT

Initial

Intermediate

Requires split, for we cannot have three entries in a single 3-node
FINAL

For a left child:
- Binary parent transformed into a 3-node
- Middle value pushed up to the parent to become its second entry
- Smallest value in insertion node remains the parent’s left child
- Largest value in insertion node becomes the parent’s middle child

Mirrored when inserting into a right child
INSERTING IN A 3-NODE WITH A 3-NODE PARENT

Initial

First Intermediate
INSERTING IN A 3-NODE WITH A 3-NODE PARENT

Note:
- Moved the middle item in the insertion node to the parent and split the remaining entries into binary nodes
- Cannot have 3 entries in 3-node, so this requires a split

Second Intermediate

15 24 48

2 22 43 78
INSERTING IN A 3-NODE WITH A 3-NODE PARENT

Final

Note:
- Every leaf node is at the same level
BTreeDemo
QUADTREES

WHAT’S IN A NAME?

Spatial records include a sense of location
  - Latitude and Longitude
  - X, Y, Z coordinates

We need a data structure capable of holding multi-dimensional keys

Must provide $O(\log n)$ insertion, removal, and search

Due to the potentially vast nature of a spatial data set, the structure should be efficient with respect to space
SPATIAL REPRESENTATION
Many different trees fall under the quadtree umbrella

- Point Quadtree
- Region Quadtree
- PR Quadtree
- MX Quadtree

All quadtrees recursively subdivide a space

Classified by:
- Type of data the tree represents
- Method of decomposition
- Resolution (fixed or variable)
APPLICATIONS

- **Image Compression:**
  Reduce the amount of space required to store certain images

- **Computer Graphics:**
  Object culling (ignore things far away). Once we know a node of the tree is not visible, then we know nothing in its subtrees is visible either

- **Terrain:**
  Store information about a particular area (e.g., terrain assets, sensor data, etc. . . .)

- **AI Pathfinding:**
  Some agents use quadtrees to select the shortest path (average weights for areas of map)

- **Collision Detection:**
  Quickly rule out objects nowhere near the collision
A basic quadtree interface using the java collection interface.

Is this the correct place to define a two-dimensional compareTo?
public class PointQuadTree<K extends Comparable<? super K>, V>
extends AbstractCollection<IQuadTree.IEntry<K,V>> implements IQuadTree<K, V> {

- The Java Collection interface includes many methods we require.
- Extending AbstractCollection in the concrete class helps simplify the development process.

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>add(E e)</td>
</tr>
<tr>
<td></td>
<td>Ensures that this collection contains the specified element (optional operation).</td>
</tr>
<tr>
<td>boolean</td>
<td>addAll(Collection&lt;? extends E&gt; c)</td>
</tr>
<tr>
<td></td>
<td>Adds all of the elements in the specified collection to this collection (optional operation).</td>
</tr>
<tr>
<td>void</td>
<td>clear()</td>
</tr>
<tr>
<td></td>
<td>Removes all of the elements from this collection (optional operation).</td>
</tr>
<tr>
<td>boolean</td>
<td>contains(Object o)</td>
</tr>
<tr>
<td></td>
<td>Returns true if this collection contains the specified element.</td>
</tr>
<tr>
<td>boolean</td>
<td>containsAll(Collection&lt;?&gt; c)</td>
</tr>
<tr>
<td></td>
<td>Returns true if this collection contains all of the elements in the specified collection.</td>
</tr>
<tr>
<td>boolean</td>
<td>equals(Object o)</td>
</tr>
<tr>
<td></td>
<td>Compares the specified object with this collection for equality.</td>
</tr>
<tr>
<td>int</td>
<td>hashCode()</td>
</tr>
<tr>
<td></td>
<td>Returns the hash code value for this collection.</td>
</tr>
<tr>
<td>boolean</td>
<td>isEmpty()</td>
</tr>
<tr>
<td></td>
<td>Returns true if this collection contains no elements.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt;</td>
<td>iterator()</td>
</tr>
<tr>
<td></td>
<td>Returns an iterator over the elements in this collection.</td>
</tr>
<tr>
<td>boolean</td>
<td>remove(Object o)</td>
</tr>
<tr>
<td></td>
<td>Removes a single instance of the specified element from this collection, if it is present (optional operation).</td>
</tr>
<tr>
<td>boolean</td>
<td>removeAll(Collection&lt;?&gt; c)</td>
</tr>
<tr>
<td></td>
<td>Removes all of this collection’s elements that are also contained in the specified collection (optional operation).</td>
</tr>
</tbody>
</table>
How do we determine which quadrant a given entry falls within? Standard Java `compareTo` only provides a one-dimensional result

- Greater, less, or equal

We need a comparison establishing

- Quadtree: one of four quadrants
- Octree: one of eight quadrants

Note: 4D requires sixteen links per node, so the concept does not scale well past the third dimension
@Override
public int compareTo2D(IEntry<? extends K1, V1> other) {

    if (equals(other)) {
        return EntryPresent;
    }
    if (isNorth(other)) {
        if (isEast(other)) {
            return NorthEast;
        }
        return NorthWest;
    }
    if (isEast(other)) {
        return SouthEast;
    }
    return SouthWest;
}
ENTRY EQUALS

```java
@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (!(obj instanceof Entry)) {
        return false;
    }
    Entry<K1, V1> other = (Entry<K1, V1>) obj;
    return posX == other.getPosX() && posY == other.getPosY();
}
```

Our entry’s test for equivalence only checks the keys, so an object with a different value but identical keys evaluates as equal
ENTRY HASHCODE

Although we have no plans to use a hash table in our implementation, we changed `.equals`, so best practice dictates we must override `.hashCode`

```java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 17;
    result = prime * result + ((posX == null) ? 0 : posX.hashCode());
    result = prime * result + ((posY == null) ? 0 : posY.hashCode());
    return result;
}
```
Irregular decomposition (i.e. quadrants may vary in size)
Point entry holds the entry and links to four children
Unique tree based on insertion order
- Selection of root may impact performance
- Degrades to $O(n)$ in the worst case

What insertion order generates this performance?
INSERTION IN A POINT QUADTREE

- Begins with a search for the quadrant within which to place the value
- If quadrant is non-null, then it must further subdivide, so recursively call the insert method on the appropriate quadrant
- If quadrant is null, then bind it to a new node containing the spatial data
Insertion order impacted the point quadtree's shape

- Toronto inserted before Buffalo
- What happens when we delete Chicago

Deletion, therefore, may significantly impact performance

Strategy:
- Find the point node to delete
- Remove the node from the tree
- Reinsert all the deleted nodes children into its parent
CASE: IMAGE COMPRESSION

- How can we reduce space required to store image?
- 1024x1024 image holds over 1M pixels
- Many cells possess identical values (low entropy)
- Early Solution:
  Instead of viewing as grid of uniformly sized pixels, collection of dynamically sized bounding boxes
  Each node stores average value of children
REGION QUADTREE (TRIE)

Allow elimination of large sections with one inspection

Successive, regular subdivision of the data into four equal-sized quadrants
  - Standard size in powers of two
  - If region is not homogeneous, it subdivides and continues

Variable resolution data structure
Collection of maximal blocks
Two node types
  1. Internal nodes hold links to children
  2. Leaves hold the data
CASE: COLLISION DETECTION

Collision Detection
Region quadtree modified for point data
Associates data points with quadrants
Insensitive to input order
All insertions result in the creation of a new leaf
Internal nodes guide toward the leaf
Two node styles
- Internal nodes only hold pointers to children
- Leaf nodes only hold values
PR QUADTREE INSERTION

- Empty: create new leaf node at root with required data
- Reached leaf: replace existing leaf with an internal node, and add the existing and new values to this node
- Process continues until only one point occupies the region identified by the key
- May add many levels with a single insertion (e.g., two points at minimal distance in same quadrant)

Must divide space such that the two closest data points in the set partition into different regions
In a PR Quadtree, all data resides in the leaves
   Removal requires elimination of leaf
   Easiest search tree case

Deletion may result in the elimination of an internal node’s only child
   Internal node thus no longer serves a purpose
   Collapse (remove) the internal node
   Continue recursively eliminating internal nodes until one reaches an internal node with multiple children
CASE: TERRAIN

Infinite, procedurally generated worlds
Generate terrain patches on demand
Need to accommodate patches of clustered data wherever the player visits
Will, by definition, only be one patch of terrain for a given latitude / longitude
Need to identify neighboring patches of terrain as players and NPCs move from one region to another