Lecture 7: BST Range Search

- k-d Trees
- Binary Space Partitioning Trees
- Nearest-Neighbor Search

BST Range Search

Instead of finding an exact match, find all items whose keys fall between a range of values, e.g., between \( m \) and \( z \), inclusive.

Example applications:

BST Range Search: Algorithm

```c
void rangesearch(Link root, Key searchrange[],
                 Key subtreerange[], List results)
{
    1. if root is in search range, add root to results
    2. compute range of left subtree
    3. if search range covers all or part of left subtree, search left
    4. compute range of right subtree
    5. if search range covers all or part of right subtree, search right
    6. return results
}
```

(Other traversal orders are also ok)

BST Range Search: Example

T:
```
    k
   /|
  d p
 /|
a f m z
```

rangeSearch(T, [m-z], [a-z], results)
(T’s range is “whole universe”)

1. is \( k \) in \([m-z]\)?
2. does \([a-j]\) overlap \([m-z]\)?
3. does \([l-z]\) overlap \([m-z]\)?
4. search \( p \)’s subtree
   a. is \( p \) in \([m-z]\)? results ← \( p \)
   b. does \([l-o]\) overlap \([m-z]\)?
      a. search \( m \)’s subtree
         i. is \( m \) in \([m-z]\)? results ← \( m \)
         b. does \([l-l]\) overlap \([m-z]\)?
            i. search \( n \)’s subtree
               a. is \( n \) in \([m-z]\)? results ← \( n \)
               b. does \([q-z]\) overlap \([m-z]\)?
                  i. search \( z \)’s subtree
                     a. is \( z \) in \([m-z]\)? results ← \( z \)
BST Range Search: Support Functions

1. if root is in search range, i.e.,
   root->key <= searchrange[MAX], and
   root->key >= searchrange[MIN]
   add node to results

2. compute subtree’s range: replace upper (lower) bound
   of left (right) subtree’s range by root->key−1 (+1)

3. if search range covers all or part of subtree’s range,
   search subtree
   • each subtree covers a range of key values
   • compute overlap between subtree’s range and search range
   • no overlap if either
     searchrange[MAX] < subtreerange[MIN] or
     searchrange[MIN] > subtreerange[MAX]

BST Range Search: Other Details

How to express range when the keys are floats?
• be careful with numerical precision and floating point
  errors [one of this week’s discussion topic]

How to support duplicate keys?
• be consistent about using ≤ or <
  • if ≤, the range for the left subtree would be closed, e.g.,
    [−∞, 0], and the range for the right subtree half open,
    e.g., (0, +∞]

Multidimensional Search

Example applications:

A k-d tree can handle all these queries with \( O(\log n) \)
insert and search times (it can also handle partial,
range, and approximate matches)

k-d Trees

A k-d tree is a binary search tree (not covered in
textbook, link to original 1975 paper on syllabus)

At each level of the k-d tree, keys from a different search
dimension is used as the discriminator
• keys for any given level are all from the same dimension indicated by
  the discriminator
• nodes on the left subtree of a node have keys with value < the
  node’s key value along this dimension
• nodes on the right subtree have keys with value > the node’s key
  value along this dimension

We cycle through the dimensions as we go down the tree
• for example, given keys consisting of x- and y-coordinates, level 0
could discriminate by the x-coordinate, level 1 by the y-coordinate,
level 2 again by the x-coordinate, etc.
**k-d Trees: Example**

Given points in a Cartesian plane on the left, a corresponding k-d tree is shown on the right.

![Diagram of a k-d tree with points and a tree structure.]

**k-d Tree Insert**

```cpp
void kdTree::
insert(Link &root, Item newitem, int disc)
{
  if (root == NULL) {
    root = new Node(newitem);
    return;
  }

  if (newitem.key[disc] < root->item.key[disc]) // or <=
    insert(root->left, newitem, (disc+1)%dim);
  else if (newitem.key[disc] > root->item.key[disc])
    insert(root->right, newitem, (disc+1)%dim);
}
```

If new item's key is smaller than root's along the dimension indicated by the discriminator, recursive call on left subtree else recursive call on right subtree.

In both cases, switch the discriminator before traversing the next level of the tree, cycling through the dimensions.

**k-d Tree Search**

Search works similarly to insert, using a discriminator to cycle through the dimensions as one recurses down the levels: $O(\log n)$ time.

The tree we built was nicely balanced.

- If nodes are inserted randomly, on average we will get a balanced tree: $O(\log n)$ time.
- If nodes inserted are sorted, recursively insert the median of the range to build a balanced tree: $O(n \log n)$ time.

**k-d Tree Remove**

To remove a node on a level with discriminator along dimension $j$:

- If the node is a leaf, remove it.
- Else if node has right subtree, find the $j$-minimum node in the right subtree.
- Replace node with $j$-minimum node and repeat until you reach a leaf, then remove the leaf.
- Else find the $j$-maximum node in the left subtree, replace, repeat, remove.
- $j$-minimum means minimum along the $j$ dimension, analogously $j$-maximum.
- $O(\log n)$ time.
**k-d Tree Remove Example**

- **Delete (35,60)**

```
35,60
20,45 x
10,35
20,20 x
60,20
70,10 y
```

- **Replacement x-minimum**

```
60,20
```

- **Replacement y-maximum**

```
70,10
```

**k-d Tree Remove Example**

- **Delete (50,30)**

```
50,30
20,45 x
10,35
20,20 x
60,20 y
70,10 y
```

- **Replacement y-maximum**

```
60,20
```

- **Replacement x-minimum**

```
70,10
```

**k-d Tree Remove Example**

- **Delete (60,20)**

```
60,20
20,45 x
10,35
20,20 x
50,30 y
```

- **Replacement x-minimum**

```
60,20
```

- **Leaf node, can be deleted**

```
70,10
```

**k-d Tree Remove Summary**

- **Delete (35,60)**

```
35,60
20,45 x
10,35
20,20 x
60,20 y
70,10 y
```

- **Replacement x-minimum**

```
60,20
```

- **Leaf node, can be deleted**

```
70,10
```
Multidimensional Range Search

Example applications:

What would the `kdTree::rangedsearch()` function signature be?
- What are its formal arguments?
- What would it return?

---

`k-d Tree Range Search`

```cpp
def void kdTree::rangedsearch(Link root, int disc, 
    Key searchrange[], 
    Key subtreerange[], List results)
```

- cycle through the dimensions as we traverse down the tree, same as with `kdTree::insert()`
- `searchrange[]` holds 2 values (min, max) per dimension
- subtree’s ranges are defined for all dimensions
  - for 2D, both `searchrange[]` and `subtreerange[]` define a rectangle
  - for dimension j’s range, `subtreerange[2*j]` holds the lower bound, `subtreerange[2*j+1]` holds the upper bound
    - these need to be updated as we go down the levels

---

Example: Binary Space Partitioning (BSP) Trees

**Axis-aligned BSP tree**: a k-d tree used for spatial range search, with the following distinctions:
- items are stored in leaf nodes only
- other internal nodes store only the coordinates used to partition space
- an item that spans multiple partitions are stored in all corresponding leaf nodes

Example usages:
- occlusion culling and collision detection in computer graphics, computer games, robot motion planning

---

Axis-Aligned BSP Tree: Idea

Splitting plane aligned to x, y, or z axis

- Minimal box
- Split along plane
- Split along plane
- Split along plane

Tomas Akenine-Moller © 2002
Axis-Aligned BSP Tree: Build

Each internal node holds a divider plane
Leaves hold geometry

Example Use: Collision Detection

Blue wants to get to object $e$
Blue draws a straight line from itself to $e$
Now it needs to know which objects in the room could potentially obstruct it if it follows this straight line
What is the brute force way of finding potentially obstructing objects?

Collision Detection with BSP

Algorithm:
- use a plane to divide space into 2 parts
- objects whose $j$-coordinate is smaller than that of the partition go to the left subtree
- objects with $j$-coordinate larger than the partition’s go to the right subtree
- repeat for the two partitions, cycling through the coordinates
Collision Detection with BSP

Blue searches the BSP and found that at $x=30$, $e$ is to its right, so it can ignore objects $a, b, g, h, i, j, k$.

At $y=30$, $e$ is to its left, so it can ignore objects $l, n$.

It now knows that it would potentially have to navigate around $d, f, c$, and $m$.

Of $d, f, c$, and $m$, $c$ and $m$ are in the same area as $e$, compute intersection with them first.

Collision Detection with BSP

Blue needs to find out if any object is in its way to $(52, 45)$.

What objects does Blue need to compute intersection with?

Blue is in the left side of the $x=52$ partition, and only object $d$ is on the same side.

So Blue only needs to compute intersection with $d$.

Collision Detection with BSP

Now Blue is at $(52, 45)$, it faces away from $y=45$, so it needs to turn around to go to $e$.

To get to $e$, it needs to find if there are other objects in its way.

Since it is now in the same region as $e$, it only needs to compute intersection with other objects in the region: $c$ and $m$.
Collision Detection with BSP

Compared to the brute force method, we only need to compute intersection with $c$, $m$, $d$, and $c$ and $m$ again: 5 intersection computations instead of 3*13 intersections

Under normal $k$-d trees, we search both subtrees when the search range spans both subtrees

Under BSP trees however, items that span subtrees are split into two halves such that only one subtree needs to be searched

Fixed-Range Search

The range searches we have considered all have fixed ranges specified in world space, e.g.,

- of the whole alphabet space, search in range $[m–z]$
- of the whole Cartesian space, search in range $[(-\infty, 6), (-\infty, 8)]$
- find all (or $k$) departures within 2 hours of noon
  - relative to local point of reference
  - but can be transformed into a fixed- and absolute-range query in world-space, i.e., [10 am to 2 pm]

Nearest-Neighbor Search

In nearest-neighbor search the range is not pre-specified and is in local space, e.g.,

- find the ATM machine nearest to me
- find the $k$ restaurants nearest to me

[Known as the $k$-nearest neighbor ($k$-NN) problem]

Using $k$-d tree to support nearest-neighbor queries in 2D:

- not the most efficient solution in theory
- but everyone uses it in practice
- time complexity: $O(n^{1/2} + k)$

Nearest Neighbor Search: Algorithm

1. start with a reference point ($\mathbf{X}$) and an infinite search range
2. if root node matches query and the distance from reference point to root is smaller than current search range, reduce search range to this distance
3. determine whether the reference point is to the left or right of root (along the discriminating coordinate)
4. recursively search the branch of the tree where the reference point belongs
5. upon returning from the recursive search, the search range may have been further reduced, check to see if the current search range covers the other branch of the tree that has not been searched
6. if so, recursively search the other branch of the tree; otherwise prune the other branch
7. return results
Nearest Neighbor Search

void nnsearch(Link root, int disc, Key reference, double *searchradius, Key subtreerange[], List results)

Assume all data points match query

Reference is to the left of A, search left child of A first
Found a node (B) closer to reference than A
Reduce search range to distance(reference, B)
Returning back to $A$, current (new) search range **no longer overlaps** $A$'s right child, prune the right child, return result.

Since the root doesn't hold a data point, the search range remains at $\infty$ as we search the right side of the $k$-d tree where the reference point is.

In this example, the $k$-d tree is formed by splitting planes, similar to an axis-aligned BSP, hence the internal nodes do not always hold data points.

As we traverse to the second level of the $k$-d tree, we still cannot reduce the search range.
We've reached a leaf node and found two data points that match our query (assume all data points do), reduce the search range to the minimum distance between the reference point and the two data points.

Returning back to the parent node, we found that the search range overlaps the range of the right subtree, search the right subtree.

Found a closer neighbor in the left branch of the right subtree, further reduce the search range.

Using current search range and each subtree's range, prune parts of the tree that could NOT include the nearest neighbor.
Nearest Neighbor Search

Using current search range and each subtree’s range, prune parts of the tree that could NOT include the nearest neighbor.

k Nearest Neighbors Search

To find k nearest neighbors, maintain k current best instead of just the best [different k from k-d tree]. Branches are only eliminated when they can't hold any neighbor closer than any of the k current best.

Programming Assignment 2

Topic: Location-Based Services

“Consumer and advertiser expenditure on location-based services (LBS) to approach $10B by 2016, with search advertising accounting for just over 50%.” LA Times, 6/10/11

Due date: Thu, 10/13, 10:00 pm

To be done individually (no group or team)

No STL (iostream and string are allowed, they are part of the C++ standard library, not part of STL)

Geographic Coordinate System

A point on earth is given by its latitude and longitude

Latitude 0° is at the Equator
Latitude 90° is at the North Pole
Latitude −90° is at the South Pole

Longitude 0° is the Prime Meridian
Longitude ±180° is a the International Dateline (more or less)
**Standard Coordinate Frame**

The World Geodetic System 84 represents the earth as an ellipsoid with the origin located at the Earth’s center of mass (±2 cm).

WGS84 is the current standard coordinate frame used by GPS systems.

**Lat/Lon of Some Cities on Earth**

<table>
<thead>
<tr>
<th>City</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kinshasa</td>
<td>–4.325</td>
<td>15.322222</td>
</tr>
<tr>
<td>Istanbul</td>
<td>41.01224</td>
<td>28.976018</td>
</tr>
<tr>
<td>Mumbai</td>
<td>18.975</td>
<td>72.825833</td>
</tr>
<tr>
<td>Jakarta</td>
<td>–6.2</td>
<td>106.8</td>
</tr>
<tr>
<td>Shanghai</td>
<td>31.2</td>
<td>121.5</td>
</tr>
<tr>
<td>Queenstown</td>
<td>–45.031111</td>
<td>168.6625</td>
</tr>
<tr>
<td>Honolulu</td>
<td>21.30889</td>
<td>–157.826111</td>
</tr>
<tr>
<td>San Francisco</td>
<td>37.7793</td>
<td>–122.4192</td>
</tr>
<tr>
<td>Ann Arbor</td>
<td>42.281389</td>
<td>–83.748333</td>
</tr>
<tr>
<td>Rio de Janeiro</td>
<td>–22.908333</td>
<td>–43.196389</td>
</tr>
<tr>
<td>Reykjavik</td>
<td>64.133333</td>
<td>–21.933333</td>
</tr>
<tr>
<td>Casablanca</td>
<td>33.533333</td>
<td>–7.583333</td>
</tr>
</tbody>
</table>

**Problem Specification**

Read in a “database” of location data.
Each entry consists of:

- **lat** lon name tag

where the **lat** and **lon** are floats and name and tag are single words.

To simplify the assignment, we limit acceptable latitude to between 0° and 90° and we limit acceptable longitude to between 0° and –180° (covering North America).

**Sample Location Database**

In one column:

<table>
<thead>
<tr>
<th>lat</th>
<th>lon</th>
<th>name tag</th>
<th>lat</th>
<th>lon</th>
<th>name tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>42.2893</td>
<td>–83.7391</td>
<td>NorthsideGrill restaurant</td>
<td>42.2984</td>
<td>–83.7195</td>
<td>Qdoba fastfood</td>
</tr>
<tr>
<td>42.3036</td>
<td>–83.7090</td>
<td>UMCU bank</td>
<td>42.2780</td>
<td>–83.7409</td>
<td>Ashley’s pub</td>
</tr>
<tr>
<td>42.2831</td>
<td>–83.7485</td>
<td>TheBrokenEgg restaurant</td>
<td>42.2984</td>
<td>–83.7195</td>
<td>Panera bakery</td>
</tr>
<tr>
<td>42.2892</td>
<td>–83.7200</td>
<td>GreatPlainsBurger fastfood</td>
<td>42.2846</td>
<td>–83.7451</td>
<td>Zingerman’s restaurant</td>
</tr>
<tr>
<td>42.3033</td>
<td>–83.7053</td>
<td>Evergreen restaurant</td>
<td>42.2797</td>
<td>–83.7496</td>
<td>WestendGrill restaurant</td>
</tr>
<tr>
<td>42.2785</td>
<td>–83.7413</td>
<td>CometCoffee cafe</td>
<td>42.2808</td>
<td>–83.7486</td>
<td>KaiGarden restaurant</td>
</tr>
<tr>
<td>42.2845</td>
<td>–83.7463</td>
<td>Yamato restaurant</td>
<td>42.2909</td>
<td>–83.7178</td>
<td>UMCU bank</td>
</tr>
<tr>
<td>42.2806</td>
<td>–83.7497</td>
<td>GrizzlyPeak pub</td>
<td>42.2806</td>
<td>–83.7493</td>
<td>CafeZola restaurant</td>
</tr>
<tr>
<td>42.2810</td>
<td>–83.7486</td>
<td>Vinology restaurant</td>
<td>42.3047</td>
<td>–83.7090</td>
<td>Kroger supermarket</td>
</tr>
<tr>
<td>42.3030</td>
<td>–83.7066</td>
<td>TCF bank</td>
<td>42.2830</td>
<td>–83.7467</td>
<td>NoThai fastfood</td>
</tr>
<tr>
<td>42.2803</td>
<td>–83.7479</td>
<td>ArborBrewingCompany pub</td>
<td>42.3048</td>
<td>–83.7083</td>
<td>AABank bank</td>
</tr>
<tr>
<td>42.2780</td>
<td>–83.7449</td>
<td>CreditUnion bank</td>
<td>42.2827</td>
<td>–83.7470</td>
<td>CafeVerde cafe</td>
</tr>
<tr>
<td>42.2804</td>
<td>–83.7497</td>
<td>Sweetwaters cafe</td>
<td>42.2780</td>
<td>–83.7449</td>
<td>UMCU bank</td>
</tr>
<tr>
<td>42.2795</td>
<td>–83.7438</td>
<td>TCF bank</td>
<td>42.2828</td>
<td>–83.7485</td>
<td>Heidelberg pub</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>42.2792</td>
<td>–83.7409</td>
<td>PotBelly fastfood</td>
</tr>
</tbody>
</table>
You can assume there will be no duplicate records (all four fields being the same) in the database.

The lat/lon may be duplicated, but as long as the name and/or the tag is different, records with the same lat/lon are not considered duplicates.

We limit the lat/lon precision to 4 decimal places ($\pm 0.0001^\circ$), which translates to about 11 m longitudinal distance at the equator and about 5.56 m longitudinal distance at latitude 60°.

The database is followed by a single blank line and one or more queries.

There are three types of queries (not ordered):

1. exact-match query, led by an ‘@’ sign:
   @ 42.2982 –83.7200
   @ 42.2984 –83.7195
   @ 42.2980 –83.7109

2. range-match query, led by an ‘r’:
   r 42.2806 –83.7493 0.0004
   r 42.2812 –83.7521 0.002

3. nearest neighbor query, led by an ‘n’ sign:
   n 42.2785 –83.7461 bank
   n 42.2785 –83.7461 bookstore
   n 42.3033 –83.7078 bank
   n 42.3034 –83.7078 bank

The two floats following the ‘@’ and ‘r’ signs should be obvious.

The third float following the ‘r’ sign specifies the search range from the provided location, in degree.

We limit the range to be at most 0.5 degree from the provided location, so in total the search range is limited to 1 degree (enough to cover the New York Metropolitan Area or the Tokyo/Yokohama megalopolis, the two largest urban conglomerations in the world, by area).
Problem Specification

Given the acceptable lat/lon and the limit on search range, the acceptable latitude on a range search is limited to 0.5° to 89.5° and the acceptable longitude is limited to –0.5° to –179.5°

No need to translate degree to linear units!

Problem Specification

The tag following the lat/lon in nearest location search is the queried tag, i.e., a record matches the query only if the record contains the same tag as the queried tag

To simplify implementation, we will use a “bounding box” covering the search radius to perform nearest neighbor search

void rangesearch(Link root, int disc, Key reference, double *searchradius, Key bbox[], Key subtreerange[], List results)

Bounding box search range covers the circle whose radius is the distance between the reference point and the nearest neighbor

distance(A, B) = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}

Bounding box is shrunk as a nearer neighbor is found

A data point is considered a “neighbor” only if its tag matches that of the queried tag
Output

Given the above database and queries, the output of your program should be:

- 42.2982 -83.7200 GreatPlainsBurger fastfood
- 42.2984 -83.7195 Panera bakery
- 42.2984 -83.7195 Qdoba fastfood

No record found

- 42.2984 -83.7195 Panera bakery
- 42.2984 -83.7195 Qdoba fastfood

No record found

- 42.2806 -83.7493 CafeZola restaurant
- 42.2806 -83.7497 GrizzlyPeak pub
- 42.2804 -83.7497 Sweetwaters cafe

- 42.2806 -83.7493 CafeZola restaurant
- 42.2806 -83.7497 GrizzlyPeak pub
- 42.2804 -83.7497 Sweetwaters cafe

- 42.2806 -83.7493 CafeZola restaurant
- 42.2806 -83.7497 GrizzlyPeak pub
- 42.2804 -83.7497 Sweetwaters cafe

- 42.2806 -83.7493 CafeZola restaurant
- 42.2806 -83.7497 GrizzlyPeak pub
- 42.2804 -83.7497 Sweetwaters cafe

Location-Based Search

Finding exact matches:
- must be implemented using hashing
- figure out: what hash function to use
- figure out: how to resolve collisions
- assume your program will be used all over the acceptable region

Finding range and nearest neighbor matches:
- must be implemented using k-d tree
- search range forms a rectangle/bounding box
- no need to implement node removal
- may assume database is not sorted

k-d Tree

PA2 Grading Criteria

Working, efficient solution (75%):
- autograder will use –O3 compile flag for timing, so make sure your Makefile also uses the –O3 flag

Test cases (20%)

Code is readable, well-documented (5%):
- pay attention to PA1 grade report and avoid being penalized for the same stylistic issues
Files Organization

How would you organize your code into files?

Alternative 1: main.cpp (NOT)
Alternative 2: main.cpp, hash.h, hash.cpp, kdtree.h, kdtree.cpp, array.h, array.cpp, linkedlist.h linkedlist.cpp, location.h, location.cpp (NOT)
Alternative 3: lbs281.cpp, adts.h, hash.h, kdtree.h, location.h, location.cpp

Your choice would be different, but try not to split it up into too many files!

Time Requirements

How long does it take to do PA2?

<table>
<thead>
<tr>
<th>Task</th>
<th>Lines of Code</th>
<th>% Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>design (and writing spec)</td>
<td>n/a (2 days)</td>
<td></td>
</tr>
<tr>
<td>parse input (incl. unit test)</td>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>hashing (incl. unit test)</td>
<td>132</td>
<td>27</td>
</tr>
<tr>
<td>kdtree insert</td>
<td>88</td>
<td>7</td>
</tr>
<tr>
<td>kdtree range search</td>
<td>142</td>
<td>23</td>
</tr>
<tr>
<td>kdtree nearest neighbor</td>
<td>130</td>
<td>38</td>
</tr>
<tr>
<td>whole globe (excl. nn)</td>
<td>102</td>
<td>+28</td>
</tr>
</tbody>
</table>