CSE 326: Data Structures
Binary Search Trees

Today’s Outline

- Dictionary ADT / Search ADT
- Quick Tree Review
- Binary Search Trees

ADTs Seen So Far

- Stack
  - Push
  - Pop
- Queue
  - Enqueue
  - Dequeue

Then there is decreaseKey…

The Dictionary ADT

- Data:
  - a set of (key, value) pairs
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

The Dictionary ADT is also called the “Map ADT”

Phony
Peter
Henry
James
CSE 666
CSE 002
CSE 002

A Modest Few Uses

- Sets
- Dictionaries
- Networks
  : Router tables
- Operating systems
  : Page tables
- Compilers
  : Symbol tables

Probably the most widely used ADT!

Implementations

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked-list</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Unsorted array</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Sorted array</td>
<td>log n + n</td>
<td>Θ(log n)</td>
<td>log n + n</td>
</tr>
</tbody>
</table>

What limits the performance?

Time to move elements, can we mimic BitSearch with BST?
Tree Calculations

Recall: height is max number of edges from root to a leaf

Find the height of the tree...

$$\text{height}(t) = 1 + \max \{\text{height}(t.\text{left}), \text{height}(t.\text{right})\}$$

runtime: $O(N)$ (constant time for each node; each node visited twice)

Tree Calculations Example

How high is this tree?

- height(B) = 1
- height(C) = 4
- so height(A) = 5

More Recursive Tree Calculations: Tree Traversals

A traversal is an order for visiting all the nodes of a tree

Three types:
- Pre-order: Root, left subtree, right subtree
- In-order: Left subtree, root, right subtree
- Post-order: Left subtree, right subtree, root

Inorder Traversal

```c
void traverse(BNode t){
    if (t != NULL)
        traverse (t.left);
    process t.element;
    traverse (t.right);
}
```

Binary Trees

- Binary tree is
  - a root
  - left subtree (maybe empty)
  - right subtree (maybe empty)

- Representation

Binary Tree: Representation
**Binary Tree: Special Cases**

- **Complete Tree**
- **Perfect Tree**
- **Full Tree**

**Binary Tree: Some Numbers!**

For binary tree of height \( h \):
- max # of leaves: \( 2^h \), for perfect tree
- max # of nodes: \( 2^{h+1} - 1 \), for perfect tree
- min # of leaves: \( 1 \), for “list” tree
- min # of nodes: \( h+1 \), for “list” tree

Average Depth for \( N \) nodes?

**Binary Search Tree Data Structure**

- **Structural property**
  - each node has \( \leq 2 \) children
  - result:
    - storage is small
    - operations are simple
    - average depth is small
- **Order property**
  - all keys in left subtree smaller than root’s key
  - all keys in right subtree larger than root’s key
  - result: easy to find any given key
- **What must I know about what I store?**

**Example and Counter-Example**

**Find in BST, Recursive**

```java
Node find(Object key, Node root) {
    if (root == NULL)
        return NULL;
    if (key < root.key)
        return find(key, root.left);
    else if (key > root.key)
        return find(key, root.right);
    else
        return root;
}
```

**Find in BST, Iterative**

```java
Node find(Object key, Node root) { 
    while (root != NULL && root.key != key) { 
        if (key < root.key) 
            root = root.left; 
        else 
            root = root.right; 
    } 
    return root; 
}
```
Insert in BST

5/30/2008 19

Insert(13)
Insert(8)
Insert(31)

Insertions happen only at the leaves – easy!

BuildTree for BST

5/30/2008 20

• Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.

Runtime depends on the order!

– in given order

$\Theta(n^2)$

– in reverse order

$\Theta(n^2)$

5, 3, 7, 2, 1, 6, 8, 9 better: $\Theta(n \log n)$

Bonus: FindMin/FindMax

5/30/2008 21

• Find minimum

• Find maximum

Deletion in BST

5/30/2008 22

Why might deletion be harder than insertion?

Lazy Deletion

5/30/2008 23

Instead of physically deleting nodes, just mark them as deleted

+ simpler
+ physical deletions done in batches
+ some adds just flip deleted flag

– extra memory for “deleted” flag
– many lazy deletions = slow finds
– some operations may have to be modified (e.g., min and max)

Non-lazy Deletion

5/30/2008 24

• Removing an item disrupts the tree structure.

• Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.

• Three cases:
  – node has no children (leaf node)
  – node has one child
  – node has two children
Non-lazy Deletion – The Leaf Case

Delete(17)

Deletion – The One Child Case

Delete(15)

Deletion – The Two Child Case

Delete(5)

What can we replace 5 with?

- succ from right subtree
- pred from left subtree

Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:

- succ from right subtree: findMin(t.right)
- pred from left subtree : findMax(t.left)

Now delete the original node containing succ or pred

- Leaf or one child case – easy!

Finally...

7 replaces 5

Original node containing 7 gets deleted

Balanced BST

Observation

- BST: the shallower the better!
- For a BST with n nodes
  - Average height is $O(\log n)$
  - Worst case height is $O(n)$
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

Solution: Require a Balance Condition that

1. ensures depth is $O(\log n)$ – strong enough!
2. is easy to maintain – not too strong!
Potential Balance Conditions
1. Left and right subtrees of the root have equal number of nodes
   - Too weak! Do height mismatch example
2. Left and right subtrees of the root have equal height
   - Too weak! Do example where there’s a left chain and a right chain, no other nodes

Potential Balance Conditions
3. Left and right subtrees of every node have equal number of nodes
   - Too strong! Only perfect trees
4. Left and right subtrees of every node have equal height
   - Too strong! Only perfect trees

CSE 326: Data Structures
AVL Trees

Balanced BST
Observation
- BST: the shallower the better!
- For a BST with \( n \) nodes
  - Average height is \( O(\log n) \)
  - Worst case height is \( O(n) \)
- Simple cases such as insert(1, 2, 3, ..., \( n \)) lead to the worst case scenario

Solution: Require a Balance Condition that
1. ensures depth is \( O(\log n) \) – strong enough!
2. is easy to maintain – not too strong!

The AVL Balance Condition
AVL balance property:

Left and right subtrees of every node have heights differing by at most 1

- Ensures small depth
  - Will prove this by showing that an AVL tree of height \( h \) must have a lot of (i.e. \( O(2^h) \)) nodes
- Easy to maintain
  - Using single and double rotations
The AVL Tree Data Structure

Structural properties
1. Binary tree property
   (0, 1, or 2 children)
2. Heights of left and right subtrees of every node
differ by at most 1
Result:
Worst case depth of any node is: $O(\log n)$

Ordering property
– Same as for BST

Proving Shallowness Bound

Let $S(h)$ be the min # of nodes in an AVL tree of height $h$

Trees of height $h = 1, 2, 3, \ldots$

Claim: $S(h) = S(h-1) + S(h-2) + 1$

Solution of recurrence: $S(h) = O(2^h)$
(like Fibonacci numbers)

An AVL Tree

AVL trees: find, insert

- **AVL find:**
  – same as BST find.
- **AVL insert:**
  – same as BST insert, except may need to “fix” the AVL tree after inserting new value.
AVL tree insert
Let \( x \) be the node where an imbalance occurs.

Four cases to consider.  The insertion is in the
1. left subtree of the left child of \( x \).
2. right subtree of the left child of \( x \).
3. left subtree of the right child of \( x \).
4. right subtree of the right child of \( x \).

**Idea:** Cases 1 & 4 are solved by a **single rotation**.
Cases 2 & 3 are solved by a **double rotation**.

---

**Bad Case #1**

```
Insert(6)  
Insert(3)  
Insert(1)  
```

Where is AVL property violated?

---

**Fix: Apply Single Rotation**

AVL Property violated at this node (x)

![Diagram](image)

Single Rotation:
1. Rotate between \( x \) and child

---

**Bad Case #2**

```
Insert(1)  
Insert(6)  
Insert(3)  
```

---

**Fix: Apply Double Rotation**

AVL Property violated at this node (x)

![Diagram](image)

Double Rotation
1. Rotate between \( x \)’s child and grandchild
2. Rotate between \( x \) and \( x \)’s new child
Double rotation in general

\[ h \geq 0 \]

\[ W < h < X < c < Y < a < Z \]

Height of tree before? Height of tree after? Effect on Ancestors?

Double rotation, step 1

Double rotation, step 2

Imbalance at node X

Single Rotation
1. Rotate between x and child

Double Rotation
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child

Single and Double Rotations:

Inserting what integer values would cause the tree to need a:
1. single rotation?
2. double rotation?
3. no rotation?

Insertion into AVL tree

1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:
   • case #1: Perform single rotation and exit
   • case #2: Perform double rotation and exit

Both rotations keep the subtree height unchanged.
Hence only one rotation is sufficient!
Easy Insert

Insert(3)

Unbalanced? No

Hard Insert (Bad Case #1)

Insert(33)

Unbalanced? Yes, at 15

How to fix?
- Single rotate
- Zig-zig

Unbalanced? No

How to fix?
- Single rotate

Single Rotation

Hard Insert (Bad Case #2)

Insert(18)

Unbalanced? Yes, at 15

How to fix?
- Double rotate
- Zig-zag

Single Rotation (oops!)

Double Rotation (Step #1)

Still unbalanced.
But like zig-zig tree!
AVL Trees Revisited

- **Balance condition:** Left and right subtrees of every node have heights differing by at most 1
  - Strong enough: Worst case depth is $O(\log n)$
  - Easy to maintain: one single or double rotation

- Guaranteed $O(\log n)$ running time for
  - Find?
  - Insert?
  - Delete?
  - buildTree?

$\Theta(n \log n)$

CSE 326: Data Structures
Splay Trees

Single and Double Rotations

AVL Trees Revisited

- What extra info did we maintain in each node?
- Where were rotations performed?
- How did we locate this node?
Other Possibilities?

- Could use different balance conditions, different ways to maintain balance, different guarantees on running time, …
- Why aren’t AVL trees perfect? Extra info, complex logic to detect imbalance, recursive bottom-up implementation
- Many other balanced BST data structures
  - Red-Black trees
  - AA trees
  - Splay Trees
  - 2-3 Trees
  - B-Trees
  - …

Splay Trees

- Blind adjusting version of AVL trees – Why worry about balances? Just rotate anyway!
- Amortized time per operation is $O(\log n)$
- Worst case time per operation is $O(n)$ – But guaranteed to happen rarely

Insert/Find always rotate node to the root!

Recall: Amortized Complexity

If a sequence of $M$ operations takes $O(M f(n))$ time, we say the amortized runtime is $O(f(n))$.

- Worst case time per operation can still be large, say $O(n)$
- Worst case time for any sequence of $M$ operations is $O(M f(n))$

Average time per operation for any sequence is $O(f(n))$

Amortized complexity is worst-case guarantee over sequences of operations.

Recall: Amortized Complexity

- Is amortized guarantee any weaker than worstcase? Yes, it is only for sequences
- Is amortized guarantee any stronger than averagecase? Yes, guarantees no bad sequences
- Is average case guarantee good enough in practice? No, adversarial input, bad day, …
- Is amortized guarantee good enough in practice? Yes, again, no bad sequences

The Splay Tree Idea

If you’re forced to make a really deep access:

Since you’re down there anyway, fix up a lot of deep nodes!

Find/Insert in Splay Trees

1. Find or insert a node $k$
2. Splay $k$ to the root using: zig-zag, zig-zig, or plain old zig rotation

Why could this be good??

1. Helps the new root, $k$
   - Great if $k$ is accessed again
2. And helps many others!
   - Great if many others on the path are accessed
Splaying node $k$ to the root: Need to be careful!

One option (that we won’t use) is to repeatedly use AVL single rotation until $k$ becomes the root: (see Section 4.5.1 for details)

What’s bad about this process? Is pushed almost as low as $k$ was. Bad seq: find($k$), find($r$), find(...), ...

Splay: Zig-Zag*

*Just like an... Helps those in blue

AVL double rotation

Hurts those in red

Which nodes improve depth? $k$ and its original children

Splay: Zig-Zig*

*Is this just two AVL single rotations in a row? Not quite – we rotate $g$ and $p$, then $p$ and $k$

Why does this help? Same number of nodes helped as hurt. But later rotations help the whole subtree.

Special Case for Root: Zig

Relative depth of $p$, $Y$, $Z$? Relative depth of everyone else?

Down 1 level Much better

Why not drop zig-zig and just zig all the way? Zig only helps one child!

Splaying Example: Find(6)

Think of as if created by inserting 6,5,4,3,2,1 – each took constant time – a LOT of
Still Splaying 6

Finally…

Another Splay: Find(4)

Example Splayed Out

But Wait…
What happened here?

Didn’t two find operations take linear time instead of logarithmic?

What about the amortized $O(\log n)$ guarantee?

Why Splaying Helps

- If a node $n$ on the access path is at depth $d$ before the splay, it’s at about depth $d/2$ after the splay
- Overall, nodes which are low on the access path tend to move closer to the root
- Splaying gets amortized $O(\log n)$ performance. (Maybe not now, but soon, and for the rest of the operations.)
**Practical Benefit of Splaying**

- No heights to maintain, no imbalance to check for
  - Less storage per node, easier to code

- Data accessed once, is often soon accessed again
  - Splaying does implicit *caching* by bringing it to the root

**Splay Operations: Find**

- Find the node in normal BST manner
- Splay the node to the root
  - if node not found, splay what would have been its parent

**What if we didn’t splay?**

*Amortized guarantee fails!*

Bad sequence: find(leaf $k$), find($k$), find($k$), …

**Splay Operations: Insert**

- Insert the node in normal BST manner
- Splay the node to the root

**What if we didn’t splay?**

*Amortized guarantee fails!*

Bad sequence: insert($k$), find($k$), find($k$), …

**Splay Operations: Remove**

*Everything else splayed, so we’d better do that for remove*

**Join(L, R):**

- Given two trees such that (stuff in L) < (stuff in R), merge them:
  - Splay max

**Join**

- Does this work to join any two trees?
  - No, need L < R

**Delete Example**

- Similar to BST delete – find max = find element with no right child
- Then attach R

**Delete(4)**

- Find max
- Then attach R

- Does this work to join any two trees?
  - No, need L < R
Splay Tree Summary

- All operations are in amortized $O(\log n)$ time
- Splaying can be done top-down; this may be better because:
  - only one pass
  - no recursion or parent pointers necessary
  - we didn’t cover top-down in class
- Splay trees are very effective search trees
  - Relatively simple
  - No extra fields required
  - Excellent locality properties:
    - frequently accessed keys are cheap to find

Splay

B-Trees

Weiss Sec. 4.7

CSE 326: Data Structures

B-Trees

Weiss Sec. 4.7

Time to access (conservative)
1 ns per instruction

Disk (mechanical device)
- Seek time + Transfer time
  - Once seek, then transfer
- Slow: many GB

Memory
- Main Memory
  - 40-100 ns
- Cache
  - 2-10 ns
- CPU (has registers)
  - 1 ns per instruction

Summary: Accesses to memory are painful, especially slower levels
Trees so far

- BST
  - Up to 2 children, height can be N-1
  - Operations WC: O(N)
  - 10 Mil disk accesses
  
  - AVL
    - Up to 2 children, height can be log N
    - Operations WC: O(log N)
    - 23 disk accesses
    - Simpler implementation than AVL
    - Operations WC: O(N)
    - Amortized: O(log N)

- Splay
  - Up to 2 children, height can be N-1
  - Operations WC: O(N)
  - Amortized: O(log N)

Dictionary ADT: Find, Insert, Remove

Trees from book: N = 10 Million
drivers in WA state
Want to do even better, make

\[ M = \begin{cases} 5, & \text{10 accesses} \\ 10, & \text{7 accesses} \\ 100, & \text{3-4 accesses} \end{cases} \]

AVL trees

Suppose we have 100 million items

\( (100,000,000) \):

\[ \log_2 100,000,000 = 26.6 \]

\[ \log_{128} 100,000,000 = 3.8 \]

\[ \text{Wow!} \]

M = 5, 10 accesses; M=10, 7 accesses; M=100, 3-4 accesses

M = 5
- Up to 2 children, height can be N-1
- Operations WC: O(N)
- 10 Mil disk accesses

M = 10
- Up to 2 children, height can be log N
- Operations WC: O(log N)
- 23 disk accesses

M = 100
- Up to 2 children, height can be N-1
- Simpler implementation than AVL
- Operations WC: O(N)
- Amortized: O(log N)

\[ \text{Draw picture of tree and memory and show} \]
\[ \text{that each access can be an access to disk} \]

Example from book: N = 10 Million
drivers in WA state

Want to do even better, make

\[ M = \begin{cases} 5, & \text{10 accesses} \\ 10, & \text{7 accesses} \\ 100, & \text{3-4 accesses} \end{cases} \]

\[ M = 5 \]
- Up to 2 children, height can be N-1
- Operations WC: O(N)
- 10 Mil disk accesses

\[ M = 10 \]
- Up to 2 children, height can be log N
- Operations WC: O(log N)
- 23 disk accesses

\[ M = 100 \]
- Up to 2 children, height can be N-1
- Simpler implementation than AVL
- Operations WC: O(N)
- Amortized: O(log N)

\[ \text{Amortized: O(log N)} \]

\[ \text{Draw picture of tree and memory and show} \]
\[ \text{that each access can be an access to disk} \]

Example from book: N = 10 Million
drivers in WA state

Want to do even better, make

\[ M = \begin{cases} 5, & \text{10 accesses} \\ 10, & \text{7 accesses} \\ 100, & \text{3-4 accesses} \end{cases} \]

\[ M = 5 \]
- Up to 2 children, height can be N-1
- Operations WC: O(N)
- 10 Mil disk accesses

\[ M = 10 \]
- Up to 2 children, height can be log N
- Operations WC: O(log N)
- 23 disk accesses

\[ M = 100 \]
- Up to 2 children, height can be N-1
- Simpler implementation than AVL
- Operations WC: O(N)
- Amortized: O(log N)

\[ \text{Amortized: O(log N)} \]

\[ \text{Draw picture of tree and memory and show} \]
\[ \text{that each access can be an access to disk} \]

Example from book: N = 10 Million
drivers in WA state

Want to do even better, make

\[ M = \begin{cases} 5, & \text{10 accesses} \\ 10, & \text{7 accesses} \\ 100, & \text{3-4 accesses} \end{cases} \]

\[ M = 5 \]
- Up to 2 children, height can be N-1
- Operations WC: O(N)
- 10 Mil disk accesses

\[ M = 10 \]
- Up to 2 children, height can be log N
- Operations WC: O(log N)
- 23 disk accesses

\[ M = 100 \]
- Up to 2 children, height can be N-1
- Simpler implementation than AVL
- Operations WC: O(N)
- Amortized: O(log N)

\[ \text{Amortized: O(log N)} \]

\[ \text{Draw picture of tree and memory and show} \]
\[ \text{that each access can be an access to disk} \]

Example from book: N = 10 Million
drivers in WA state

Want to do even better, make

\[ M = \begin{cases} 5, & \text{10 accesses} \\ 10, & \text{7 accesses} \\ 100, & \text{3-4 accesses} \end{cases} \]

\[ M = 5 \]
- Up to 2 children, height can be N-1
- Operations WC: O(N)
- 10 Mil disk accesses

\[ M = 10 \]
- Up to 2 children, height can be log N
- Operations WC: O(log N)
- 23 disk accesses

\[ M = 100 \]
- Up to 2 children, height can be N-1
- Simpler implementation than AVL
- Operations WC: O(N)
- Amortized: O(log N)

\[ \text{Amortized: O(log N)} \]

\[ \text{Draw picture of tree and memory and show} \]
\[ \text{that each access can be an access to disk} \]

Example from book: N = 10 Million
drivers in WA state

Want to do even better, make

\[ M = \begin{cases} 5, & \text{10 accesses} \\ 10, & \text{7 accesses} \\ 100, & \text{3-4 accesses} \end{cases} \]

\[ M = 5 \]
- Up to 2 children, height can be N-1
- Operations WC: O(N)
- 10 Mil disk accesses

\[ M = 10 \]
- Up to 2 children, height can be log N
- Operations WC: O(log N)
- 23 disk accesses

\[ M = 100 \]
- Up to 2 children, height can be N-1
- Simpler implementation than AVL
- Operations WC: O(N)
- Amortized: O(log N)

\[ \text{Amortized: O(log N)} \]

\[ \text{Draw picture of tree and memory and show} \]
\[ \text{that each access can be an access to disk} \]
B-Tree Properties

- Data is stored at the leaves. Leaves are at least \( \frac{L}{2} \) full, kind of like complete tree.
- All leaves are at the same depth and contains between \( \lceil L/2 \rceil \) and \( L \) data items.
- Internal nodes store up to \( M-1 \) keys.
- Internal nodes have between \( \lceil M/2 \rceil \) and \( M \) children.
- Root (special case) has between 2 and \( M \) children (or root could be a leaf).

Tree about \( \log n \) deep. Actually, \( \log_{M/2} n \).

Example Again

B-Tree with \( M = 4 \) and \( L = 4 \)

Find(33, 2, 16)
Insert(18, 16, 71)
Insert(80) (split into 2 leaves each w. min # values) [put 71 at top]

Building a B-Tree

[Diagram showing building a B-Tree]

Splitting the Root

[Diagram showing splitting the root]

Overflowing leaves

[Diagram showing overflowing leaves]

Propagating Splits

[Diagram showing propagating splits]
**Insertion Algorithm**

1. Insert the key in its leaf
2. If the leaf ends up with \( L+1 \) items, overflow!
   - Split the leaf into two nodes:
     - original with \( \lceil (L+1)/2 \rceil \) items
     - new one with \( \lfloor (L+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
3. If an internal node ends up with \( M+1 \) items, overflow!
   - Split the node into two nodes:
     - original with \( \lceil (M+1)/2 \rceil \) items
     - new one with \( \lfloor (M+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
4. Split an overflowed root in two and hang the new nodes under a new root

**After More Routine Inserts**

- Insert(89)
- Insert(79)

**Deletion**

1. Delete item from leaf
2. Update keys of ancestors if necessary

**Deletion and Adoption**

A leaf has too few keys!
- So, borrow from a sibling
- Sibling pointer (and parent pointers)
- What if not enough to borrow from?

**Does Adoption Always Work?**

- What if the sibling doesn’t have enough for you to borrow from?
  - e.g. you have \( \lceil L/2 \rceil - 1 \) and sibling has \( \lceil L/2 \rceil \) ?

**Deletion and Merging**

A leaf has too few keys!
- So, delete the leaf
Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than \( \lceil \frac{N}{2} \rceil \) items, **underflow**!
   - Adopt data from a sibling; update the parent;
   - If adopting won’t work, delete node and merge with neighbor;
   - If the parent ends up with fewer than \( \lceil \frac{N}{2} \rceil \) items, **underflow**!

3. If an internal node ends up with fewer than \( \lceil \frac{N}{2} \rceil \) items, **underflow**!
   - Adopt from a neighbor;
   - If adoption won’t work, merge with neighbor;
   - If the parent ends up with fewer than \( \lceil \frac{N}{2} \rceil \) items, **underflow**!

4. If the root ends up with only one child, make the child the new root of the tree.

   This reduces the height of the tree!
Thinking about B-Trees

- B-Tree insertion can cause (expensive) splitting and propagation
- B-Tree deletion can cause (cheap) adoption or (expensive) deletion, merging and propagation
- Propagation is rare if $M$ and $L$ are large
  (Why?)
- If $M = L = 128$, then a B-Tree of height 4 will store at least 30,000,000 items

Tree Names You Might Encounter

FYI:
- B-Trees with $M = 3, L = x$ are called 2-3 trees
  - Nodes can have 2 or 3 keys
- B-Trees with $M = 4, L = x$ are called 2-3-4 trees
  - Nodes can have 2, 3, or 4 keys

K-D Trees and Quad Trees

Range Queries

- Think of a range query.
  - “Give me all customers aged 45-55.”
  - “Give me all accounts worth $5m to $15m”
- Can be done in time ________.
- What if we want both:
  - “Give me all customers aged 45-55 with accounts worth between $5m and $15m.”

Geometric Data Structures

- Organization of points, lines, planes, etc in support of faster processing
- Applications
  - Map information
  - Graphics - computing object intersections
  - Data compression - nearest neighbor search
  - Decision Trees - machine learning

k-d Trees

- Jon Bentley, 1975, while an undergraduate
- Tree used to store spatial data.
  - Nearest neighbor search.
  - Range queries.
  - Fast look-up
- k-d tree are guaranteed $\log_2 n$ depth where $n$ is the number of points in the set.
  - Traditionally, k-d trees store points in $d$-dimensional space which are equivalent to vectors in $d$-dimensional space.
**Range Queries**

- Rectangular query
- Circular query

**Nearest Neighbor Search**

- Nearest neighbor is e.

**k-d Tree Construction**

- If there is just one point, form a leaf with that point.
- Otherwise, divide the points in half by a line perpendicular to one of the axes.
- Recursively construct k-d trees for the two sets of points.
- Division strategies
  - divide points perpendicular to the axis with widest spread.
  - divide in a round-robin fashion (book does it this way)

**k-d Tree Construction**

- divide perpendicular to the widest spread.

**k-d Tree Construction (18)**

**2-d Tree Decomposition**
**k-d Tree Splitting**

- Sorted points in each dimension
- \[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \]
- \[ a \ b \ c \ d \ e \ f \ g \ h \ i \]
- Max spread is the max of \( l_i - a_i\) and \( r_i - a_i\).
- In the selected dimension the middle point in the list splits the data.
- To build the sorted lists for the other dimensions scan the sorted list adding each point to one of two sorted lists.

**k-d Tree Construction Complexity**

- First sort the points in each dimension.
  - \( O(dn \log n) \) time and dn storage.
  - These are stored in \( A[1..d,1..n] \)
- Finding the widest spread and equally divide into two subsets can be done in \( O(dn) \) time.
- We have the recurrence
  - \( T(n,d) \leq 2T(n/2,d) + O(dn) \)
- Constructing the k-d tree can be done in \( O(dn \log n) \) and dn storage

**Node Structure for k-d Trees**

- A node has 5 fields
  - Axis (splitting axis)
  - Value (splitting value)
  - Left (left subtree)
  - Right (right subtree)
  - Point (holds a point if left and right children are null)

**Rectangular Range Query**

- Recursively search every cell that intersects the rectangle.
Rectangular Range Query

```c
print_range(xlow, xhigh, ylow, yhigh :integer, root: node pointer) {
    Case {
        root = null: return;
        root.left = null:
        if xlow < root.point.x and root.point.x < xhigh
            and ylow < root.point.y and root.point.y < yhigh
            then print(root);
        else
            if(root.axis = "x" and xlow < root.value ) or
               (root.axis = "y" and ylow < root.value ) then
                print_range(xlow, xhigh, ylow, yhigh, root.left);
            if (root.axis = "x" and xlow > root.value ) or
                (root.axis = "y" and ylow > root.value ) then
                print_range(xlow, xhigh, ylow, yhigh, root.right);
    }
}
```

k-d Tree Nearest Neighbor Search

- Search recursively to find the point in the same cell as the query.
- On the return search each subtree where a closer point than the one you already know about might be found.

Notes on k-d NNS

- Has been shown to run in O(log n) average time per search in a reasonable model.
- Storage for the k-d tree is O(n).
- Preprocessing time is O(n log n) assuming d is a constant.
Worst-Case for Nearest Neighbor Search

- Half of the points visited for a query
- Worst case $O(n)$
- But: on average (and in practice) nearest neighbor queries are $O(\log N)$

Quad Trees

- Space Partitioning

A Bad Case

Notes on Quad Trees

- Number of nodes is $O(n(1+\log(\Delta/n)))$
  where $n$ is the number of points and $\Delta$ is the ratio of the width (or height) of the key space and the smallest distance between two points
- Height of the tree is $O(\log n + \log \Delta)$

K-D vs Quad

- k-D Trees
  - Density balanced trees
  - Height of the tree is $O(\log n)$ with batch insertion
  - Good choice for high dimension
  - Supports insert, find, nearest neighbor, range queries
- Quad Trees
  - Space partitioning tree
  - May not be balanced
  - Not a good choice for high dimension
  - Supports insert, delete, find, nearest neighbor, range queries

Geometric Data Structures

- Geometric data structures are common.
- The k-d tree is one of the simplest.
  - Nearest neighbor search
  - Range queries
- Other data structures used for
  - 3-d graphics models
  - Physical simulations